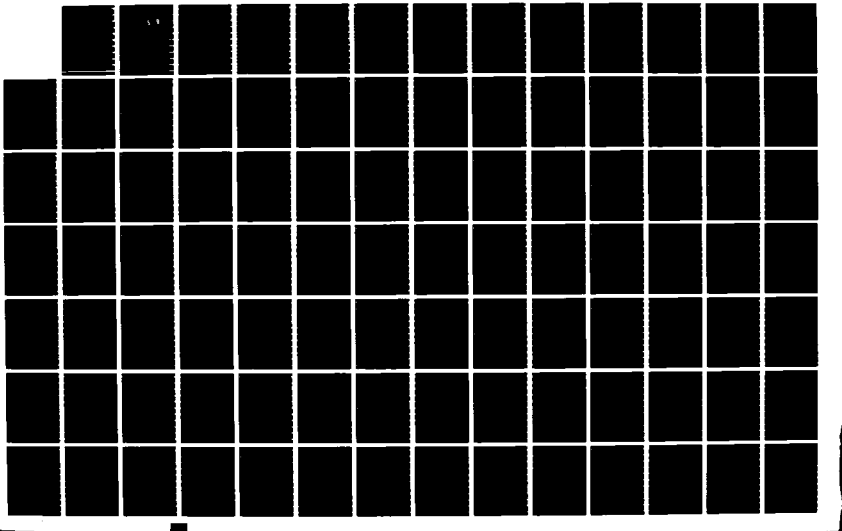
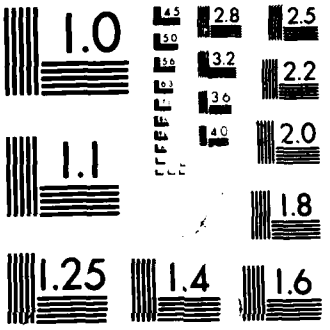
AD-A171 803 COMPILER DIRECTED MEMORY MANAGEMENT FOR NUMERICAL PROGRAMS(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB M I MALKAWI AUG 86 UILU-ENG-86-2229 1/2

UNCLASSIFIED N00014-84-C-0149 F/G 9/2 NL





August 1986

UILU-ENG-86-2229
CSG-54

(12)

COORDINATED SCIENCE LABORATORY
College of Engineering

COMPILER DIRECTED MEMORY MANAGEMENT FOR NUMERICAL PROGRAMS

Mohammad Isam Malkawi

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

AD-A171 803

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: N/A	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): UILU-ENG-86-2229 (CSG-54)	5. MONITORING ORGANIZATION REPORT NUMBER(S): none
6a. NAME OF PERFORMING ORGANIZATION: Coordinated Science Laboratory, University of Illinois; 6b. OFFICE SYMBOL (If applicable):	7a. NAME OF MONITORING ORGANIZATION: Joint Services Electronics Program
6c. ADDRESS (City, State and ZIP Code): 1101 W. Springfield Avenue, Urbana, IL 61801	7b. ADDRESS (City, State and ZIP Code): Office of Naval Research, 800 N. Quincy, Arlington, VA 22209
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Joint Services Electronics Program; 8b. OFFICE SYMBOL (If applicable): N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-84-C-0149
8c. ADDRESS (City, State and ZIP Code): Office of Naval Research, 800 Quincy, Arlington, VA 22209	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
N/A	N/A	N/A	N/A

11. TITLE (Include Security Classification): Compiler Directed Memory Management for Numerical Programs

12. PERSONAL AUTHOR(S): Mohammad I. Malkawi

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Technical	FROM ___ TO ___	August, 1986	

16. SUPPLEMENTARY NOTATION: N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Virtual Memory, Compiler Directive, Memory Directive, ALLOCATE, LOCK, UNLOCK

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report presents a new approach to the management of memory hierarchies in the multiprogramming virtual memory system. Memory management related problems are solved partially at compile time, where memory directives are inserted into the object code of a compiled program. The main objectives of memory directives are to determine the memory requirements of a program at compile time and to pass this information to the operating system at execution time. A multiprogramming system has been simulated to evaluate the performance of a compiler directed memory management policy (CD). Empirical results obtained from this study show that CD can be superior to the best known implementable policies. In particular, CD has been compared with the working set policy (WS). The results reported in this report show that CD outperforms WS by a relatively large margin.

Although CD has been designed to improve the behavior of numerical programs in virtual memory systems, it could be extended to cover other application programs. Moreover, CD has the potential of being applied to multiprocessor systems.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS ☐	Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
		none

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

unclassified
SECURITY CLASSIFICATION OF THIS PAGE

COMPILER DIRECTED MEMORY MANAGEMENT FOR NUMERICAL PROGRAMS

BY

MOHAMMAD ISAM MALKAWI

Dipl., Tashkent Polytechnical Institute, 1980
M. Eng., Yarmouk University, 1983

THESIS

Submitted in partial fulfillment of the requirement
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1986

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

JUNE 1986

WE HEREBY RECOMMEND THAT THE THESIS BY

MOHAMMAD ISAM MALKAWI

ENTITLED COMPILER DIRECTED MEMORY MANAGEMENT FOR NUMERICAL PROGRAMS

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Director of Thesis Research

Head of Department

Committee on Final Examination†

Chairman

† Required for doctor's degree but not for master's.

COMPILER DIRECTED MEMORY MANAGEMENT FOR NUMERICAL PROGRAMS

Mohammad Isam Malkawi, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1986
Janak H. Patel, Advisor

This thesis presents a new approach to the management of memory hierarchies in the multiprogramming virtual memory system. Memory management related problems are solved partially at compile time, where memory directives are inserted into the object code of a compiled program. The main objectives of memory directives are to determine the memory requirements of a program at compile time and to pass this information to the operating system at execution time. A multiprogramming system has been simulated to evaluate the performance of a compiler directed memory management policy (CD). Empirical results obtained from this study show that CD can be superior to the best known implementable policies. In particular, CD has been compared with the working set policy (WS). The results reported in this thesis show that CD outperforms WS by a relatively large margin.

Although CD has been designed to improve the behavior of numerical programs in virtual memory systems, it could be extended to cover other application programs. Moreover, CD has the potential of being applied to multiprocessor systems.

to my parents

ACKNOWLEDGEMENTS

I am deeply grateful to my parents who continue to provide me with moral and material support without which I could not have reached this stage. My wife's patience, love and care had a great impact on my work's performance and I am very grateful to her.

I would like to thank my thesis advisor Janak Patel, for his support and encouragement. Many thanks to all my friends who provided an atmosphere of brotherhood during my stay at the university campus.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Motivation and Research Objectives

Virtual memory systems VM have been around for the past few decades and continue to provide cost effective memory management despite the modern achievements in memory technology. Today modern computers ranging from supercomputers to supermicrocomputers and workstations implement virtual memory [37]. The great amount of research in the area of VM has produced several models of program behavior and several memory management policies MMP. Carr in his Ph.D. thesis [13] presents a survey of models of program behavior. Denning [20] cites two forms of program behavior models: models of programs' memory demand and models of memory management policies. Batson and Madison [30] define another model of program behavior, phase transition model, based on locality characteristics. However, the best model of program behavior, Carr concludes [13], is the program itself. In a simulation environment, as is the case of this study, a program is represented by its address reference string.

Memory management policies, cited in the literature or implemented in real systems, have been classified into two classes: the class of variable allocation, dynamic, memory management policies and the class of fixed allocation, static, memory management policies. Examples of dynamic policies are the Working Set policy (WS) [18] and its variation; the Page Fault Frequency algorithm (PFF) [14]; and globally implemented policies. Examples of static policies are Least Recently Used (LRU) and First In First Out (FIFO).

Dynamic policies have been shown to outperform static ones [10], [16]. However, they have their own problems. WS, for example, is too expensive to implement; furthermore, it is unable to avoid heavy faulting rate during interlocality transitions [23]. The Damped Working Set (DWS) [36] was introduced to avoid interlocality transition faults. However, Graham [25] showed that

DWS outperforms WS by less than 10%. The Sampled Working Set (SWS) [34] is a cheaper realization of WS, but has a poorer performance [20]. Ferrari and Yih [23] combined SWS and DWS and introduced the Variable Sampled Working Set (VSWS). VSWS performance is no worse than that of WS [23].

The page fault frequency algorithm is cheaper to implement [15] but has poorer performance than WS [25]; also, it exhibits anomalous behavior [24]. Also, WS exhibits some types of anomalies when tested against numerical programs [4], [8]. Other types of WS anomalies are discovered in a multiprogramming environment. (See Chapter 2.) Carr [13] compared WS with "global" CLOCK. The WS policy was shown to be only slightly superior to CLOCK. Carr combined the features of WS and CLOCK into a new algorithm WSclock which has a similar to WS performance, although cheaper to implement.

Based on the survey of the research published in VM area, two observations could be made. The first is on the nature of the experiments. Simulation of single programs is a common characteristic of a vast majority of the experiments, regardless of the fact that multiprogramming systems are the real VM environment. See, for example, the experiments in [3], [4], [6], [8], [16], [17], [21], [25], [28], and [36]. One can only guess that researchers have assumed that results obtained from simulating in a single programming environment would not differ significantly when applied to multiprogramming systems. One objective of this thesis is to investigate the accuracy of this claim.

The second observation is in regard to memory management policies. A common characteristic of all existing policies, whether static as is LRU or dynamic as is WS, or prefetching [39] or non-prefetching, is that they try to estimate program behavior at run time. In other words, these policies solve all memory management related problems at run time. Three memory management related problems are: 1) when to bring a page into memory, 2) which page to replace and 3) how much memory to allocate. In this thesis, this type of MMP is referred to as *run time* policies. An alternative approach to run time policies is to have some or all of memory management related

problems solved at compile time. Memory management policies using this approach will be referred to as *compiler directed* policies (CD). The main objective of this thesis is to construct and develop a compiler directed policy.

Run time policies suffer from two major drawbacks. First, the design of these policies did not take into account that program behavior varies from one program category to another. For example, numerical programs behave differently from system programs [3], [27]. Also, data base referencing has a different behavior from other types of applications [40], [38]. The second drawback results from the fact that run time policies do not consider the interaction of programs in a multiprogramming system. Programs affect each other through swapping for local policies and through paging for global policies. Also, in a multiprogramming system, the amount of free memory on the system is variable; it varies according to the load on the system and to the amount of memory occupied by each process in the system.

In this thesis, a compiler directed (CD) policy is designed with three main features:

(1) It exploits source level information at compile time. This information is passed to the operating system through *memory directives and is* used to define memory requirements of a program during execution.

(2) It is designed to respond to the changes in program intrinsic memory requirements, and to the requirements of other programs running in the system.

(3) The compiler directed policy recognizes the difference in program behavior exhibited by different program applications. It is designed specifically for numerical programs.

1.2. Overview of This Work

This work is concerned with designing a compiler directed policy. The performance of CD is evaluated in a multiprogramming environment and compared with WS, since all other run time policies either perform worse than WS or nearly the same [13], [15], [20], [23].

Chapter 2 focuses on the performance of WS in a multiprogramming environment. The working set policy is shown to exhibit anomaly types which may not be discovered in a uniprogramming environment. A multiprogramming model is used in Chapter 2 to evaluate the performance of WS. The model generates other results than those needed for the investigation of anomalies. These results are used later in Chapter 4.

Chapter 2 demonstrates how the results obtained from simulating in a multiprogramming environment may differ from those obtained from a single programming environment.

In Chapter 3 CD, a compiler directed policy, is presented. CD uses three types of directives. These directives are used by the operating system (OS) to define a process's memory requirements. We develop algorithms to be used by a preprocessor at compile time to generate memory directives. We also present algorithms for processing a directive when executed by the CPU.

Chapter 3 also deals with implementation issues of CD. In particular, a swapping strategy is developed for CD. The strategy is based on the amount of free memory available on the system and the overcommitment of memory to one or more processes in the system.

Subroutine and procedure call handling can cause a significant problem for generation of compile time directives and processing of run time directives, a problem commonly encountered in compilation techniques. Chapter 3 presents a technique for solving such problems. Issues related to the cost of CD, specially the cost associated with executing memory directives, are discussed in Chapter 3.

Performance evaluation and measurements are presented in Chapter 4. The performance of CD is compared to the performance of WS. Empirical results are gathered from a trace driven simulator of a multiprogramming system.

The conclusions drawn from this research are presented in Chapter 5 together with some suggestions for future research in this area.

CHAPTER 2

WORKING SET PERFORMANCE IN MULTIPROGRAMMING SYSTEMS

2.1. Introduction

The Working Set policy (WS) [18] is a local variable memory management policy. WS is described as follows. Let P be the set of all pages of a program. Also, let a reference string consists of a sequence of T references, $r(1), r(2) \ldots r(t) \ldots r(T)$, in which r(t) is the segment that contains the virtual address generated by a given program. Time is measured in virtual time. At virtual time t, the program's working set $W(t,\tau)$ is the subset of P which has been referenced in the previous τ virtual time units, where τ is the WS window size. The size of the working set $w(t,\tau)$ is given by the number of pages in the working set at time t. The average working set size $X(\tau)$ is defined as

$$X(\tau) = \frac{\sum_{i=1}^{T} w(t_i,\tau)}{T} \qquad (2\text{-}1)$$

where T is the length of the reference string. More definitions will be given as we proceed in this chapter.

A mechanism equivalent to the one designed by Morris [32] is used in this study to compute the working sets of a program. A reference register is associated with each page frame which is set to zero each time the page is referenced. At the same time, the reference register of every other page is incremented by one. In [32] the register is incremented at regular time intervals, rather than at every reference to a virtual address; the value in the register is an approximation to the amount of virtual time since the last reference. In our model, the value in the register is the exact amount of virtual time since the last reference. Therefore, our model computes the *exact* working sets of a program. When the value in the register equals τ, the page can be removed from the working set. The working sets are computed by performing *WS scans* of each task at each virtual

time unit t ; an approximation of the working sets is achieved by performing WS scans at various virtual time intervals [13]. In a WS scan, each page p in P is examined. If the value α in the reference register of p is equal to or larger than τ ($\alpha \geq \tau$), then p is not in the working sets; otherwise, $p \in W(t, \tau)$. Performing a scan each time a page is referenced is very expensive. However, it is the only way to capture the real dynamic behavior of WS. The main concern, in this work, is with the performance of WS rather than with the implementation cost.

2.2. WS Load Control

The working set policy requires each process to allocate enough memory to accommodate its working sets. In a multiprogramming system, however, the working sets of a program may grow beyond the available free page frames. In such a case, the working set can not be allocated in main memory. Denning [20] provides WS with the following load control policy to guarantee that the working set of a program is allocated:

> The load control maintains an uncommitted frame pool, which is a list of available page frames, and a count K of the pool's (non-negative) size. The highest priority ready task may be activated if that task's working set size w satisfies:
>
> $$w \leq K - K_0$$
>
> where K_0 is a constant specifying the desired minimum on the pool. The purpose of K_0 is to prevent needless overhead of dealing with memory overflow shortly after a new task is activated. When a page fault occurs, the page fault handler subtracts 1 from the count K. ... If K is already 0 the page fault handler will first cause the load control to preempt a page from the lowest priority active task; this implies that the lowest priority active task may not have its working set fully resident. A deactive decision may be issued by the page fault handler if the lowest priority task has its resident set reduced to naught.

Note that WS load control has two parameters τ and K_0. If τ is small, the average working set size w of each process is small and the multiprogramming level is increased. A small τ, however, increases the fault rate of each process and can lead to thrashing. Moreover, WS, using small τ, performs much worse than CD (as will be discussed in Chapter 4). A large τ reduces the fault rate but causes the process's working set to grow and depresses the multiprogramming level.

Selecting a value for K_0 represents a trade-off between maximal use of main memory and reducing the overhead that occurs when the system becomes overcommitted. If K_0 is very large, the multiprogramming level is depressed. If K_0 is very small, a swapping will be required each

time the working set expands beyond the free pool size. In this thesis, K_0 =0. However, a free page pool is dynamically created when the working set of a process grows beyond the free pool size. In this case the resident set of a "low priority process" is turned into the free pool, as a result of swapping. Swapping the resident set of a process is a modification to Denning's suggestion to preempt a page from the "lowest priority active task." Carr [13] argued that preempting the pages of the lowest priority task, one by one, "would appear to be a mistake, since the lowest priority active process will be forced to execute with a restricted resident set and will fault often and generates a great deal of paging I//O without making much progress." In the next sub-section we discuss the swapping policy used in our model.

2.3. Swapping Strategies

Swapping is the deactivation of a process that occurs when load control detects overcommitment and directs a reduction in the multiprogramming level. A process that causes memory overcommitment is called the *swapping* process. A process whose resident set is preempted is called the *swapped* process. The mechanism which handles swapping is called the *swapping mechanism* (SM). SM has the following functions: find a candidate process for swapping and preempt its resident set. In [20], a process to be swapped out is the lowest priority process in the system. While this could be the proper choice from the policy standpoint, it is not necessarily the best from a performance point of view. Carr [13] suggested four policies to select a candidate process to swap out of memory. A swapped out process can be *the faulting process*, *the last process activated*, *the smallest process*, or *the largest process*. The usefulness of any of these policies depends on the optimizing criterion under consideration and the immediate goal to be achieved. In our model, it is assumed that all the processes in the system are of equal priority. Therefore, Denning's suggestion of the lowest priority process is not practical for our model. Also, we argue that the faulting process should not be swapped out since it has just invoked SM. When it is reactivated, it may have to invoke SM again, and thus continue to be blocked. It is very likely that the last process activated has suffered a swapping just before it has been deactivated; essentially, a

discrimination may occur against one of the processes in the system. The smallest process policy discriminates against small processes, whereas the largest process policy discriminates against large processes.

We introduce a new swapping policy based on treating every process in the system with equal priority. No process should be swapped out more than one time in a row. A process that has been swapped out can not be swapped out again until all the processes in the system have experienced the pain of being swapped out. In a system with N processes, a process swapped out at time t may become a candidate for swapping only after N swapping operations. Note that N may change its value if a new process is activated or a process completes execution and leaves the system. One way of implementing this policy is to use a *CLOCK*- like mechanism. All the processes in the system are assumed to be arranged about the circumference of a circle. The CLOCK pointer (or "hand") points at the last process swapped out by SM, and is advanced "clockwise" when SM is invoked to find the next candidate for swapping.

The resident set of a swapped out process is preempted by setting the value in the reference register of each page equal to the value of τ. The size of the preempted resident set is added to the free page frames.

Another major issue of swapping is how to reclaim the working set of a swapped out process. There are two methods for a swapped out process to reclaim its working set. *Demand paging* loads a page only when that page is referenced, whether it was or it was not a member of the process's previous working set. *Prepaging* loads a collection of pages (the prepage set) when the process is activated. The prepage set, in this context, is a process's working set or its resident set when the process was swapped out. The main advantage of prepaging is to reduce page fault interrupts. However, the working set of a process has to be carefully arranged in auxiliary memory slots when the process is swapped out. Although prepaging has intuitive appeal, many systems avoid using prepaging simply because of its added complexity. From the performance standpoint, prepaging the entire working set has the same I/O effect of demand paging each page of the work-

ing set. Prepaging eliminates some page fault interrupts, and possibly, if the working set pages are sequentially stored on disk, reduces the latency seek time for all but the first page. There are other disadvantages for prepaging cited in [13]. Above all, the working set of a process is not necessarily the same when deactivated and later when reactivated. It is very likely that a process may prepage some pages which might not be referenced in the future. Besides wasting memory, prepaging may result in extra paging and wasting paging I/O capacity. At any rate, from a performance point of view, prepaging is treated as a regular page fault with a smaller service time. Page fault service time includes page fault interrupt as well as latency seek time. The model used in this study implements the demand paging mechanism.

2.4. Previous Work

Since the early 1970's many research studies have investigated the performance of WS. For bibliography and empirical results reported on WS's performance see the paper written by Denning [20], and Abusufah and Malkawi [3], [8]. Denning summarized the results of research conducted on WS [20] and drew several important conclusions. In 1972 Chu and Opderback observed that WS generates lower space time cost than the least space time generable on the LRU policy [14]. A similar conclusion could be derived from the experiments performed by Graham and Denning [26]. Denning concludes that "the evidence available suggests that *global* CLOCK and *global* LRU do not perform as well as WS." (The word *global* is added since global policies are discussed but the statement did not explicitly mention the word *global*.) It is interesting to note, however, that in the same section of the paper Denning refers to the evidence obtained from Graham's Ph.D thesis: "Graham's data shows that LRU is normally significantly worse than WS when applied to single programs" [25]. Also, Denning notes that "there is, unfortunately, little published performance data on the CLOCK and global LRU." Evidently, WS had not been compared with global LRU and global CLOCK at the time of the conclusions made in [20].

One can easily argue with the above conclusions regarding the performance of WS. It is only natural that a dynamic local policy, when properly "tuned," performs better than a static

(local) one. However, the performance of a globally implemented static policy may or may not be worse than that of WS. Such a performance can be obtained only from measurements of a multiprogrammed system. The only relevant measurements cited in [20] were those performed by Simon [35]. However, Simon compared WS and VMIN [33] in a queuing network model. His thesis did not address the problem of comparing global and local dynamic policies. On the other hand, Carr [13] simulated global CLOCK and WS policies in a multiprogramming environment. Carr concludes that "little difference between local policies (e.g., WS) and global policies (e.g., CLOCK) has been observed in a representative system". Carr introduced a new policy, WSclock, which performs as well as WS, even though "it is much simpler than any of the other WS algorithms" [13].

Compared to the page fault frequency policy, PFF [14], Denning concludes that:

> WS and PFF, when properly "tuned" by a proper choice of their control parameters, perform nearly the same and considerably better than LRU; WS has a slight tendency to produce lower space time minima than PFF. However, PFF may display anomalies for certain programs. Moreover, the performance of PFF is much more sensitive to the choice of control parameter than is the performance of WS.

However, Abusufah, et al. [4], [8] showed that WS exhibits certain types of anomalies for a certain type of programs. Out of 30 numerical programs studied in [3], all but one displayed two types of anomalies: *parameter-real memory* and fault rate-real memory anomalies [24]. Moreover, WS displayed great sensitivity to the choice of control parameter, τ [8]. Denning concluded from the empirical studies conducted by Graham [25] and Simon [35] that "the WS policy can be run with a single global τ-value and deliver throughput typically no worse than 10 percent from optimum." Alanko, Haikala, and Kutvonen [6] concluded from their empirical results that "it is impossible to find a single global τ-value that achieves the results reported in [20]." The work done by Abusufah, Lee, Malkawi, Yeu [3], [8] shows that at least 6 values of τ are needed to run a set of 17 programs within 10 percent from optimum. It is worthwhile to mention that the sensitivity of WS or PFF to the choice of control parameter can be displayed only in a multiprogrammed system. All empirical results were generated from individual reference traces, assum-

ing a uniprogrammed system, and ignoring any interaction between the programs. For this reason we believe that the sensitivity of WS to the choice of τ has not been fully investigated, although the contradiction in the reported results in literature makes previous conclusions about finding a single τ-value optimistic.

Based on comparing WS and VMIN [33] by Simon [35], Denning concludes that "no one is likely to find a policy that improves significantly over the performance of the tuned WS policy." Such a conclusion is motivated by the fact that VMIN is an optimum unimplementable policy. Carr [13] argued that Simon's work did not provide enough evidence to support such a conclusion. Simon estimated that VMIN achieves lower space time cost than WS by less than 5 percent on the average. It is interesting to note, however, that VMIN is the optimal policy for finding the minimum page fault rate; VMIN does not find a minimal space time cost. Therefore, comparing WS with VMIN can not serve as "compelling evidence" for the WS optimality. The optimal policy is DMIN [10]. In [11], DMIN showed significant improvement over both WS and VMIN.

A common characteristic of almost all research studies on the WS performance is that they use individual virtual address traces. Even when WS is compared to a global policy (global LRU), individual programs are used in the experiments; "Graham's data shows that *global* LRU is normally significantly worse than WS when applied to single programs" [20]. Denning states that:

> The WS policy serves as a dynamic estimator of the segments (pages) currently needed by a program. The WS is defined in a program's virtual time, independently of other programs; thus, there is no danger that the load on the system can influence the measurement...

While it is true that WS defined in a program's virtual time is not affected by load on the system, in a real system the resident set of pages of a program does indeed change according to system load. To maintain the resident set equal to the working set may incur overhead in terms of more page transfers not reflected in the program's intrinsic demand. Thus it is clear that the load on the system does affect the measurement of paging activities of a program. The paging activities of the WS in a multiprogramming environment were empirically measured.

The experimental model is described in the next section. Empirical results on the WS behavior in multiprogramming systems are reported in the following sections.

2.5. Multiprogramming Model

In this thesis a simple model is used to evaluate the performance of WS in a multiprogramming system. The model is shown in Figure 2-1. The same model is used for evaluating CD (described in the next chapter); specific features related to CD will be discussed in Chapter 4. The Process Queue (PQ) is implemented as a First in First Out (FIFO) and used to hold the active processes. Each process is represented by its virtual address trace. An address trace consists of references to array elements only. Initially, all array data elements are stored in the virtual storage. All instructions, constant, and simple variables are assumed to be resident in the main memory. The reason behind this assumption is that references to arrays dominate the referencing behavior of numerical programs [4], [30]. Moreover, the virtual size of the storage containing instructions, constants, and variables is usually much smaller than that used for array structures.

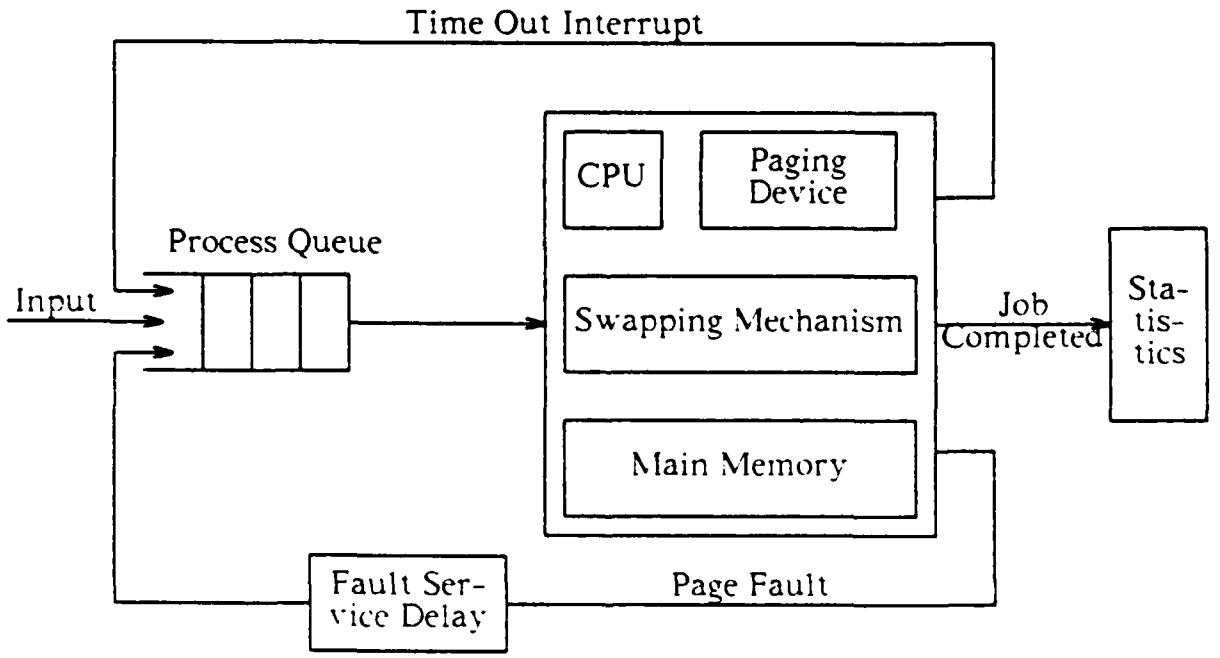


Figure 2-1: Multiprogramming model

Therefore, it would be reasonable to have the code of a program locked in main memory during the execution of that program; in case of a structured program, the code of a subprogram should be locked during the execution of that subprogram.

PQ serves as the input to the system which consists of the CPU and the main memory. The main memory is organized into a set of blocks of equal sizes *(pages)*. Similarly, the virtual storage is divided into pages of the same size. The maximum memory available on the system, θ, is used as a system variable. A list of unoccupied page frames in main memory *(free pool)* is maintained. The summation of the working sets of all programs is given by θ minus the free pool size (ρ). The main memory is initially empty. Pages of a program are paged into main memory on demand. The working set of a program is allowed to grow indefinitely into the free pool as long as the free pool size is larger than zero. If the free pool becomes empty, a swapping process is invoked and the working set of a process is removed from main memory. The pages occupied by a swapped out process are turned into the free pool. The swapping mechanism is discussed in the previous section.

A round robin scheduling strategy is used to schedule the control of the CPU by the multiple processes. A process, in control, relinquishes the CPU in one of three cases: time out interrupt, page fault occurrence, or program completion.

A time slice is used as a system variable in the model to control the time out interrupt. Upon generating a time out interrupt, the process controlling the CPU is removed from the system and entered at the tail of the PQ. However, the interrupted process's working set is not removed from main memory.

When a page fault occurs the process in control leaves the CPU and another process from the PQ gains control. The page fault is serviced by the page fault service device. The faulting process is delayed by a fault service delay element until the page fault service is completed, before it is fed back into the PQ. Page fault service time L consists of the interrupt handling time, the time spent in searching for the addressed page in the virtual storage, the transfer time of a page from disk to main memory, and the time for allocating a page frame. In this thesis we use a value of L =2000

time units: each time unit is one memory reference. The paging device is the only I/O device used in the system: this consideration further simplifies the model. In other words, the programs are assumed to be executing in a CPU bound phase. Such assumption is valid for programs which consume most of the input data at the beginning of execution and generate the output data at the end of execution. The programs used in our experiments comply with such behavior.

A process leaves the system after all of its virtual address trace has been processed. Upon completion of a process's execution, the necessary statistics are collected. These statistics include process specific and overall system statistics. The system parameters are:

(1) The maximum available physical memory on the system, θ. Very small values of θ are used for theoretical purposes. For example, θ=5 pages is clearly impractical choice of the main memory size. However, it is used to capture the behavior of WS in small memory environment, characterized by heavy swapping activity. On the other hand, using a very large θ may leads to a case similar to uniprogramming environment where the working set of a program can grow indefinitely and no swapping takes place at all. A wide range of θ values is used in order to evaluate the dependence of WS behavior on the available memory space. A large value of θ is interpreted in the context that the resident set of any program can grow to its maximum limit assuming that a program is running alone in the system.

(2) The WS parameter (the window size τ). It is difficult to find an optimal τ for any program without empirical investigation. Therefore, we vary τ from τ=1 to τ=R, where R is the reference string length of the largest program trace in the system. The window size is incremented by 5 from τ=1 to τ=1000; then τ is incremented by 100 from τ=1000 to τ=10000; beyond this value, an increment of 1000 is used. Such choices of τ are used to capture the behavior of WS in great accuracy. For small values of τ, the WS characteristics change rapidly depending on the intrinsic program behavior. In numerical programs the changes in locality structures are abrupt. The life time curves obtained from numerical programs exhibit a step-like function behavior [5]. Therefore, a very small increment in the value of τ may result in a drastic

change in the characteristics of program behavior under WS. See, for example, the life time curves reported in [7], [8].

For each program in the system, we find an optimal τ depending on the optimizing criterion. For example, we find the values of τ for which the fault rate is minimum, the space time cost is minimum, and the throughput is maximum. We also find global values of τ for which the system page faults and space time cost are minimum, and the throughput is maximum.

(3) The number of processes running simultaneously in the system. This number reflects the maximum multiprogramming level, MPL. The values of MPL used in this thesis are 3, 4, 5, and 10. However, only 5 programs are traced; the characteristics of these programs are found in Table 2-1. MPL=10 is obtained by running two copies of the same program at the same time.

(4) The context switch (CS). CS is used to control the time out interrupt. In our model, we use a large value of CS to reduce the dependence of the results on the time out interrupts. CS=1000 is much larger than the maximum possible life time between successive page faults for any of the programs: Averaging over all the programs in the system, the maximum life time is 350 time units. However, a smaller value, CS=100, is used to demonstrate the effect of CS on the paging behavior.

Process specific measures used in this chapter are:

Table 2-1
Program characteristics

Program	# Statements	# DO Stat.	# Arrays	# Array References	# Pages
MAIN	163	16	7	79,325	78
FIELD	76	9	24	10,823	60
INIT	53	14	35	10,745	174
CONDUCT	98	18	21	82,452	291
HWSCRT	135	18	7	22,721	76

(1) The average virtual resident set size $w(\theta,\tau)$. For each value of τ and θ, w is found by finding the average of the working set size of a program over its virtual execution time, R. The working set of a program is computed during each memory reference to the virtual space of the program. A page is considered in the working set if the value in its reference register (α) is less than τ. The value in the reference register is incremented during each reference. All pages with $\alpha \geq \tau$ belong to the free pool. In a uniprogramming system, the working set of a program can change only when the program is executing. In a multiprogramming system, the working set of a program is likely to be affected by other running programs. In systems using global policies, a running program's fault may result in replacing a page from another program's working set; thus, programs interact through paging. WS restricts paging activity to the program's own working set and to the free memory pool. Therefore, it seems that the working set of a program is purely intrinsic to the program behavior. We have discussed that swapping activity may, as well, be a means of interaction where the resident set of a program is affected by another program's paging activity. A swapped out process loses its entire working set in one swapping operation, or it may lose its working set pages, one by one, in several successive swapping operations if the model suggested by Denning [20] is to be used. In the previous section we discussed two methods for claiming the resident set of a process that has been swapped out of main memory. It was argued that demand paging is less complicated than prepaging. Prepaging preserves the inclusion property of the WS; namely, that $w(\tau_1) \subseteq w(\tau_2)$, where $\tau_1 < \tau_2$. The inclusion property may be violated if demand paging is used. Our model implements demand paging for the reasons discussed in the previous section. Also, with demand paging we will be able to investigate the claim that "the WS serves as a dynamic estimator of the segments (pages) currently needed by a program" [20].

(2) The page fault rate $F(\theta,\tau)$. The fault rate of a process is updated every time a reference to a nonresident page is made. The fault rate of a process depends on the intrinsic behavior of the process and on the interaction of the multiprogramming mix through swapping. Whether the demand paging or prepaging method is used, the working set of a swapped out process has to

be faulted back into main memory. In prepaging , one operation initiates an I/O for the entire working set; whereas in demand paging, a page is faulted only when a reference is made to that page.

(3) The swapping rate $S(\theta,\tau)$. S is the number of process's pages that get swapped out of main memory on the request of another process's growing working set. Swapping does not, necessarily, involve I/O operations. The working set of a process needs to be written back to the virtual storage only if the pages have been updated *(dirty pages)*. However, in this thesis we simplify the model by considering only clean pages. Therefore, the cost of swapping is associated with the swap interrupt, the search for a swapped out process, and the time for setting the values in the reference registers of the members of the working set of the swapped out process.

The overall system statistics include the system page fault rate, $F_{sys}(\theta,\tau)$ and the system average virtual memory $V_{sys}(\theta,\tau)$. F_{sys} and V_{sys} are given as the sum of the fault rates and the average virtual memory, respectively, of the individual processes.

2.6. WS Anomalies in Multiprogramming Systems

In this section we report empirical results on WS anomalies in multiprogrammed systems. Five types of anomalies are defined by Franklin, Graham and Gupta in [24]. Empirical results, reported previously, on WS anomalies have been generated from simulation of individual reference traces [4], [8] and from the analysis of individual reference strings [24]. These results show that WS exhibits two types of anomalies: namely, the *real memory-fault rate (M-F)* and *parameter-real memory (τ-M)* anomalies. $M-F$ anomaly exists if

$$M(\tau_1) < M(\tau_2) \text{ and } F(\tau_1) < F\tau_2)$$

for some values of the WS parameter τ_1 and τ_2. And $\tau-M$ anomaly exists if, for some τ_1 and τ_2,

$$\tau_1 > \tau_2 \text{ and } M(\tau_1) < M(\tau_2)$$

Both types of anomalies $M-F$ and $\tau-M$ do not violate the conditions of the generalized inclusion property proposed by Franklin et al. [24]. The other anomaly types are: *parameter-fault rate (τ-F)*

anomaly, *parameter-virtual memory (τ-V)* anomaly, and *virtual memory-fault rate (V-F)* anomaly. The WS policy can not exhibit any of these three types of anomalies when tested against individual programs in a uniprogramming environment. However, we will show that this is not the case in a multiprogramming system. We will also define two more anomaly types specific for multiprogramming systems.

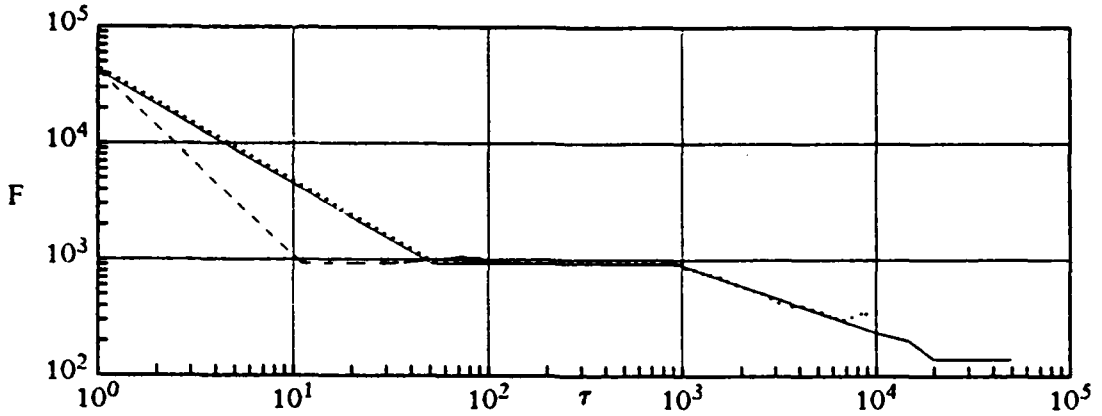
We do not report in this thesis the results on $\tau-M$ and $M-F$ anomalies since they have been empirically reported in the literature [4] [8], [24]. Besides, they have little influence on the controllability of the policy [24]. The new anomaly types discussed in this section are the *system memory-fault rate anomaly (θ-F)* and the *system memory-virtual memory anomaly (θ-V)*. These and the other anomalies are defined and discussed in details in the following subsections.

2.6.1. Parameter-fault rate anomalies

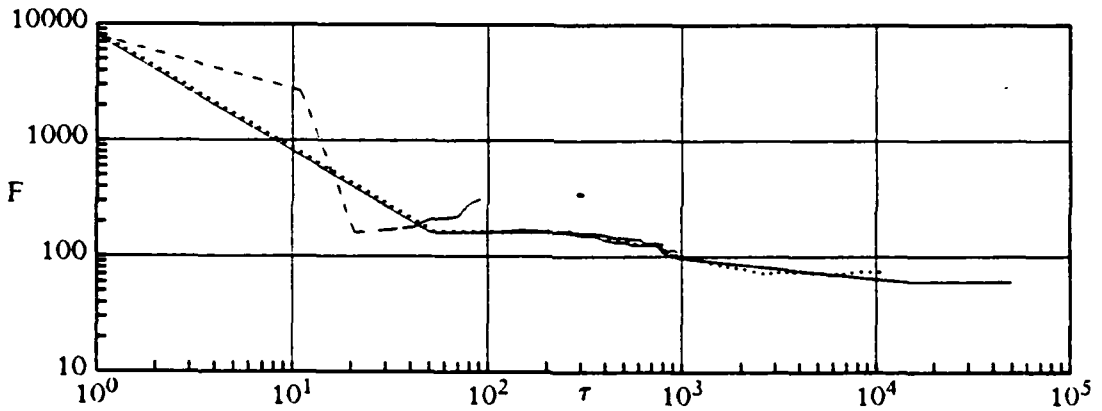
A parameter-fault rate anomaly (τ-F) in a multiprogramming system exists, for some τ_1, τ_2 and θ, if

$$\tau_1 > \tau_2 \text{ and } F(\tau_1,\theta) > F(\tau_2,\theta).$$

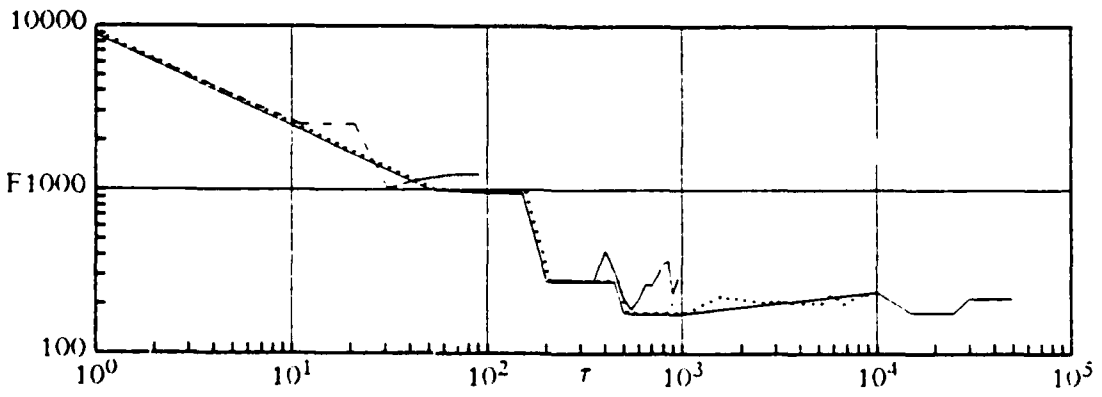
Parameter-fault rate anomalies, exhibited by individual processes are shown in Figures 2-2a - 2-2e for program MAIN, FIELD, INIT, CONDUCT, and HWSCRT respectively. Each figure contains several plots for different values of θ. We have used four different values of θ: 50, 100, 150 and 200 pages. Smaller values of θ represent the case of a high memory contention, especially for higher degrees of MPL. In each of these figures we plot the page fault rate, F, versus τ. A well behaved fault rate is a nonincreasing function of τ. An increasing portion of the curve indicates that a τ-F anomaly exists in that region. Consider, for example, Figure 2-2e for program HWSCRT for θ=200 pages (solid line). The fault rate increases from 123 to 188 when τ increases from 10,000 to 15,000. Another anomaly exists in the τ region (901,951). The anomalies reported in Figures 2-2a - 2-2e are summarized in Table 2-2. For each θ value and for each program we report the number of τ-F anomalies (N) and the size of the largest anomaly, ΔF. The anomaly size is given by $\Delta F = F(\theta,\tau_2) - F(\theta,\tau_1)$. From Figures 2-2a - 2-2e and Table 2-2 it is clear that the fault rate is



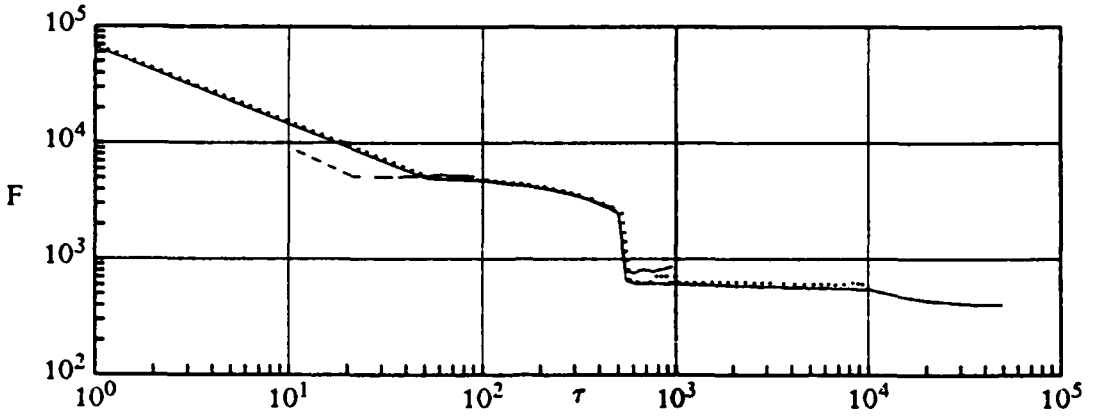
2-2a: MAIN, θ=50, 100, 150, 200



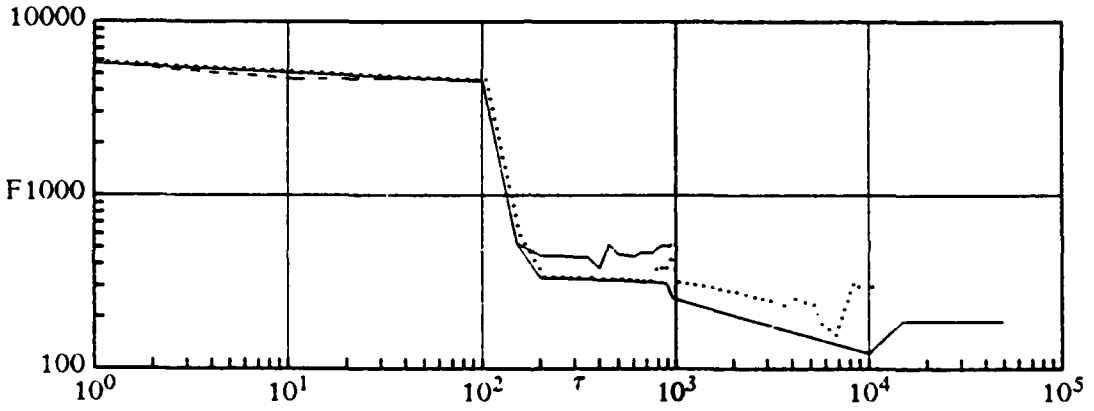
2-2b: FIELD, MPL=5, θ=50, 100, 150, 200



2-2c: INIT, MPL=5, θ=50, 100, 150, 200



2-2d: CONDUCT, MPL=5, θ=50, 100, 150, 200



2-2e: HWSCRT, MPL=5, θ=50, 100, 150, 200

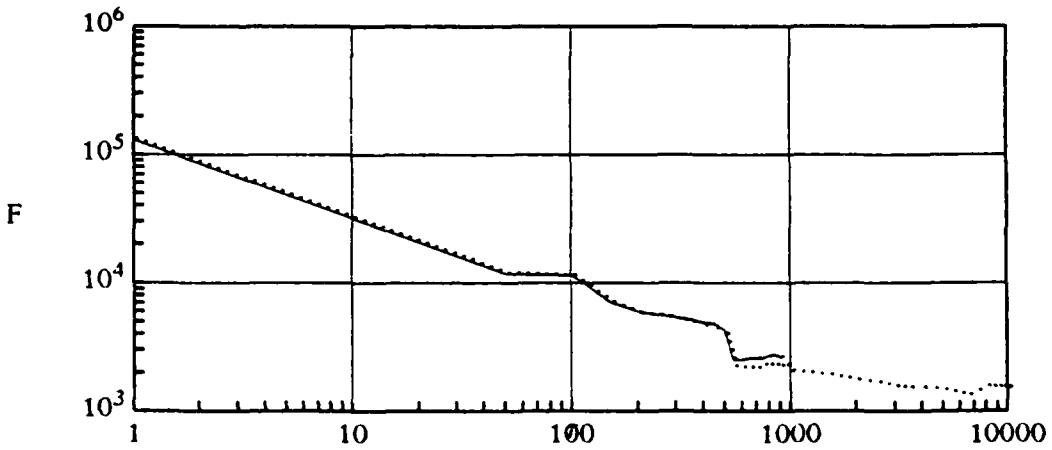
Figure 2-2: Parameter fault rate anomalies

Table 2-2
Summary of parameter fault rate anomalies

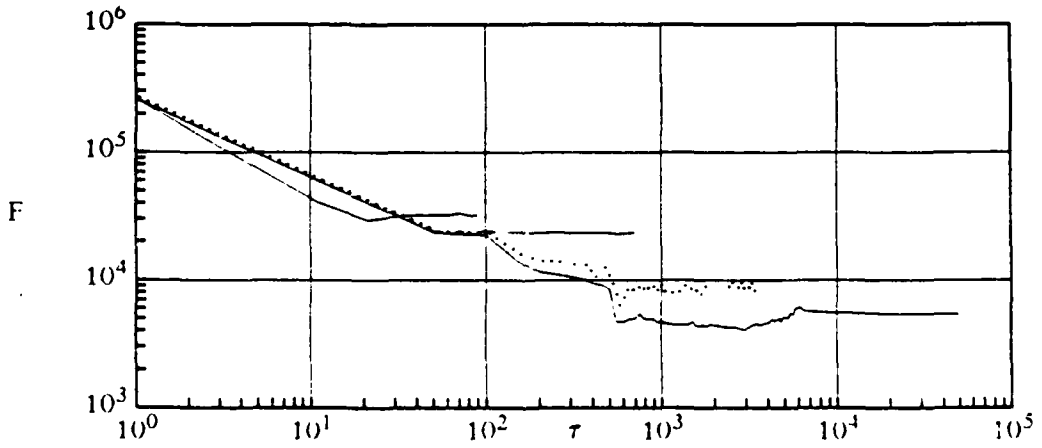
θ	MAIN		FIELD		INIT		CONDUCT		HWSCRT	
	N	ΔF	N	ΔF	N	ΔF	N	ΔF	N	ΔF
50	2	129	1	155	1	250	2	131	2	9
100	4	21	2	7	3	250	2	86	5	143
150	0	0	1	5	4	45	3	82	4	148
200	1	20	0	0	2	62	0	0	1	65

not a decreasing function of τ as in the case of well-behaved functions.

Individual program anomalies can occur in a multiprogramming system since a program's fault rate may decrease at the cost of an increase in some other program's fault rate. However, as long as the total system fault rate decreases with increasing τ, individual anomalies are not of practical importance. We would, however, like to point out that anomalies do exist even for the system fault rate. Parameter-fault rate anomalies are reported in Figures 2-3a and 2-3b, where F is the



2-3a: MPL=5, θ=100 ---, 150 ...



2-3b: MPL=10, θ= 50 --, 100 - - -, 150 ...., 200 --

Figure 2-3: System parameter fault rate anomalies

system fault rate, for MPL=5 and 10. Figure 2-3a is a plot of the system fault rate versus τ when the multiprogramming mix contains 5 programs. Two θ values are used in this plot, θ=100 (solid line) and θ=150 pages. For θ=150, anomalies exist for larger values of τ than those exhibited for θ=100 pages. The system fault rate versus τ when 10 processes are present in this system (two copies of each program) is shown in Figure 2-3b. Four plots are shown for four values of θ: θ= 50,

Table 2-3a
Parameter-fault rate anomalies (SYSTEM, MPL=10)

θ	Parameter		Fault Rate		
	τ_1	τ_2	F (θ,τ_1)	F (θ,τ_2)	ΔF
50	21	51	29332	32376	3044
	61	71	32175	33299	1124
	81	91	32038	32370	332
100	51	61	23885	24058	173
	201	251	23402	23507	105
	451	501	23367	23405	38
	551	701	22870	23459	589
150	551	951	6241	8905	2664
	1100	1300	7696	8748	1052
	1600	2400	7555	9280	1725
	2600	2700	8569	9258	689
	3000	3100	8528	9223	695
	3200	3700	8100	8870	770
	4000	4300	8388	8514	130
	4700	4800	8326	8621	285
	4900	5000	8176	8606	430
	5500	5600	8292	8425	127
	6000	6200	8178	8378	200
	7000	7100	8100	8293	193
	7600	7700	8229	8293	64
	8000	8100	7944	8025	81
200	600	700	4710	5396	686
	1200	1500	4444	4633	189
	1600	1700	4360	4420	60
	1800	1900	4345	4499	154
	2000	2100	4385	4407	22
	2200	2300	4290	4306	16
	2400	2700	4235	4276	41
	3000	5900	4055	6094	2039

100, 150, 200. For θ=50, the anomalies exist with small values of τ ($\tau < 100$). This represents the case of a high memory contention as does $\theta \leqslant 20$ for MPL=3. The anomalies demonstrated by Figures 2-3a and 2-3b are summarized in Tables 2-3a and 2-3b.

In Tables 2-3a and 2-3b we report all the anomalies exhibited at the system level for MPL=5 and MPL=10. Each anomaly region is described by two values of τ (τ_1 and τ_2) and the two corresponding values of the fault rate (F_1 and F_2). The anomaly size, ΔF, is measured as the difference between F_2 and F_1. For large values of θ (θ=150, 200) the anomalies occur with larger values of τ. Table 2-3b shows that the anomaly region for θ=100 occurs with $\tau < 551$, whereas for θ=150 it starts with $\tau > 551$.

The significance of the anomalies is emphasized by both the size and the number of anomalies. Figures 2-3 show that the anomalies do not occur in the same τ region when different θ values are used: this further complicates the control of the WS fault rate function. Furthermore, such anomalous behavior provides suitable conditions for the existence of system memory-fault rate anomalies, discussed in a later section.

2.6.2. Parameter-virtual memory anomalies

A parameter-virtual memory anomaly (τ-V) in a multiprogramming system exists for some τ_1, τ_2 and θ, if

Table 2-3b
Parameter-fault rate anomalies (SYSTEM, MPL=5)

θ	Parameter		Fault Rate		
	τ_1	τ_2	F (θ,τ_1)	F (θ,τ_2)	ΔF
100	601	651	2489	2596	107
	701	851	2594	2751	157
	901	951	2621	2717	96
150	701	751	2147	2272	125
	801	851	2269	2285	16
	901	951	2232	2243	11
	3500	4000	1495	1501	06
	6500	8000	1303	1544	241

$$\tau_1 > \tau_2 \text{ and } V(\tau_1,\theta) < V(\tau_2,\theta).$$

The anomaly size is given by: $\Delta V = V(\tau_1,\theta) - V(\tau_2,\theta)$ Figure 2-4 illustrates parameter-virtual memory anomalies for programs FIELD, INIT, and HWSCRT for MPL=5 and θ=100. It is obvious from the plots in Figure 2-4 that the average virtual memory is a nonincreasing function of τ. In Figure 2-5 V is plotted versus τ for program INIT and θ=50, 100, 150, and 200. The anomalies of Figure 2-5 are summarized in Table 2-4. Figures 2-5 and Table 2-4 show that anomalies associated

MPL=5, θ=100, FIELD —, INIT ...., HWSCRT - - -)

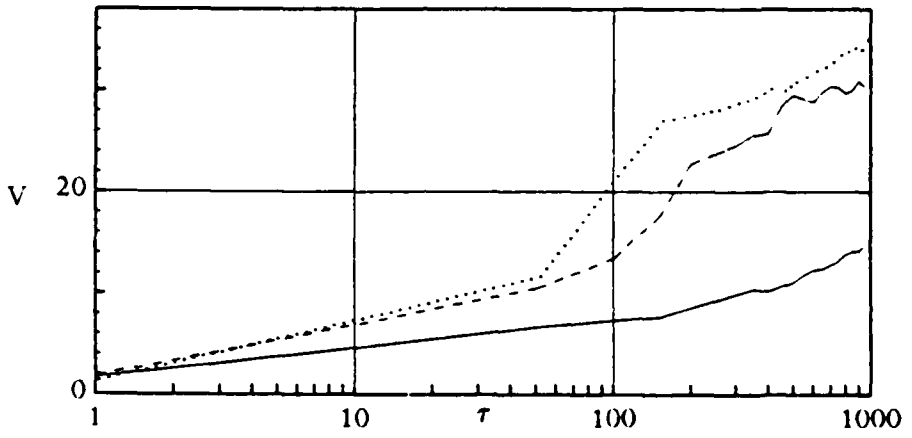


Figure 2-4: Parameter-virtual memory anomalies

MPL=5, INIT θ=50 ____ 100 - - -, 150 .... 200 ____

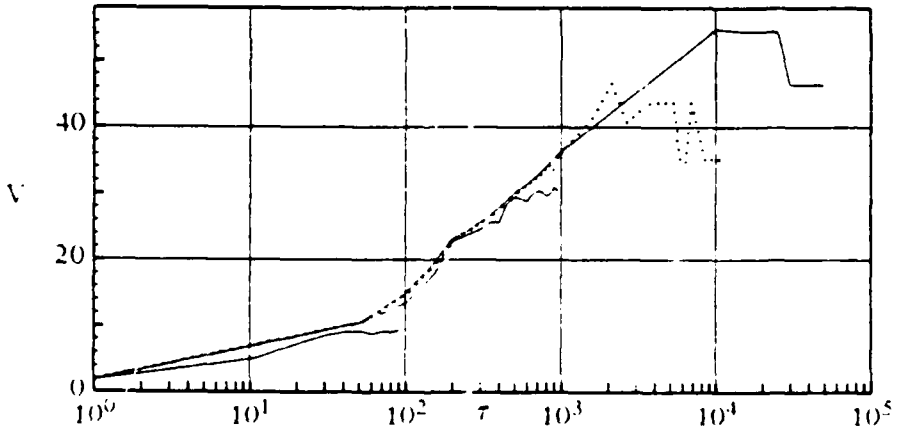


Figure 2-5: Parameter-virtual memory anomalies

with the larger values of θ tend to be shifted to the right of the anomalies associated with smaller values of θ. The τ-V anomalies when θ=200 exist for $\tau > 10{,}000$, whereas for θ=150 anomalies occur in the region $\tau < 7000$. Increasing the memory space available on the system may eliminate the anomalies in one region of τ-values and generate other anomalies in another region with larger values of τ.

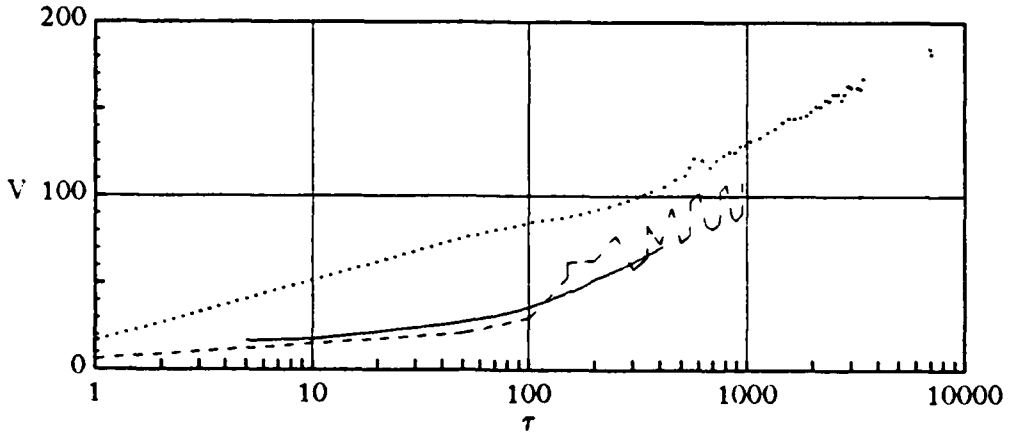
The overall system virtual size is obtained by summing up the virtual sizes of the individual processes. Figures 2-6a and 2-6b demonstrate τ-V anomalies, where V is the system's average memory, for θ= 100 and 200, respectively. Each figure contains three plots for MPL=4, 5, and 10.

The average virtual memory of a process can be reduced only as a result of a swapping process. It is very likely that a swapped out process, when reactivated, can not allocate its working set; therefore, it initiates the swapping mechanism. A chain of swapping operations will definitely lead to a reduction in the average memory space allocated to all processes. Consequently, τ-V anomalies exist at the individual process level as well as at the system level.

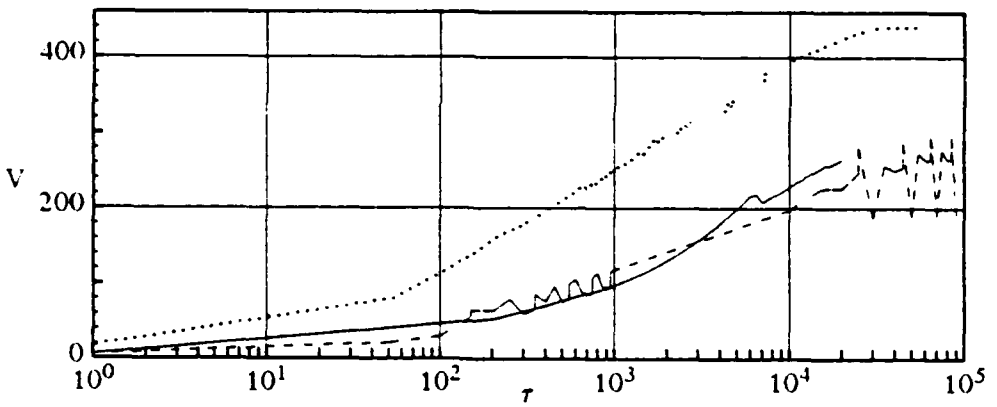
Upon reducing the average virtual memory allocated to a program or to the system, as a result of a parameter-virtual memory anomaly, the fault rate is expected to increase, assuming that the fault rate function of virtual memory is well behaved, i.e., $F(\tau_1) < F(\tau_2)$ if $V(\tau_1) > V(\tau_2)$. This suggests that a parameter-virtual memory anomaly should be associated with a parameter-

Table 2-4
Parameter-virtual memory anomalies (INIT, MPL=5)

θ	Parameter		Average Virtual Memory		
	τ_1	τ_2	$V(\theta,\tau_1)$	$V(\theta,\tau_2)$	ΔV
50	51	61	9.15	8.82	0.33
100	501	601	29.5	28.8	0.7
	701	801	30.3	29.7	0.6
150	801	851	34.0	33.4	0.6
	2000	2500	46.4	40.6	5.8
	5000	6000	43.5	34.6	8.9
	6500	7000	43.5	35.0	8.5
200	10,000	30,000	54.8	46.4	8.4



2-6a: System, θ=100, MPL=4 ---,5 - - -,10 ...)



2-6b: System, θ=200, MPL=4 ---,5 - - -,10 ...)

Figure 2-6: Parameter virtual memory anomalies

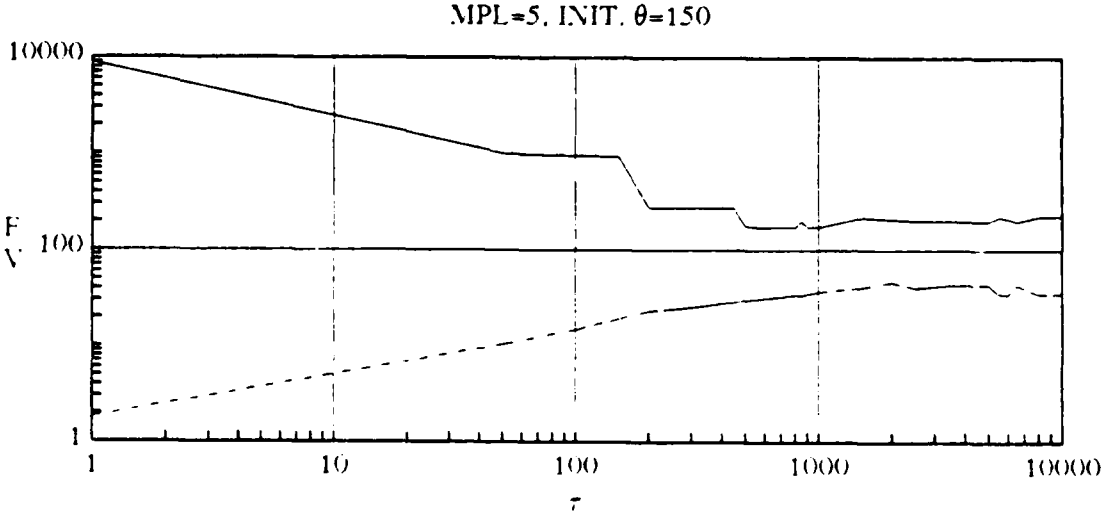


Figure 2-7: Page faults and average virtual memory versus τ

fault rate anomaly. However, the results obtained from our experiments show that this is not always the case. To illustrate this observation Figure 2-7 presnts a plot of the page fault rate and the average virtual memory versus τ for program INIT (θ=150 and MPL=5). In this figure a τ-V major anomaly occurs in the τ region [2000,2500]. The average working set size drops from 46.4 to 40.6 pages as τ increases from 2000 to 2500. In the same region, the fault rate drops from 220 to 209. The reduction in the average working set size in this region did not generate extra page faults. However, τ-V anomalies in the regions τ= [5000,6000] and [6500,7000] are accompanied with τ-F anomalies in the same regions. The fault rate increases from 198 to 226 as the average working set size is reduced from 43.5 to 34.6 pages, when τ is increased from τ=6500 to τ=7000. For θ=200, a τ-V anomaly (see Table 2-4) is not associated with a τ-F anomaly. Therefore, a parameter-fault rate anomaly does not always accompany a parameter-virtual memory anomaly.

Similar observations are made when the average working sets of all of the processes (V_{sys}) are used instead of one process. In Figure 2-8 we plot V_{sys} and F_{sys} versus τ for MPL=5 and θ=100. Six τ-V anomalies are exhibited by the figure, four of which are not matched with τ-F anomalies. For example, V_{sys} drops from 77.5 to 57.8 pages (ΔV=20) as τ increases from τ= 250 to τ=300. In the same region F_{sys} drops from 5627 to 5343 (ΔF=284).

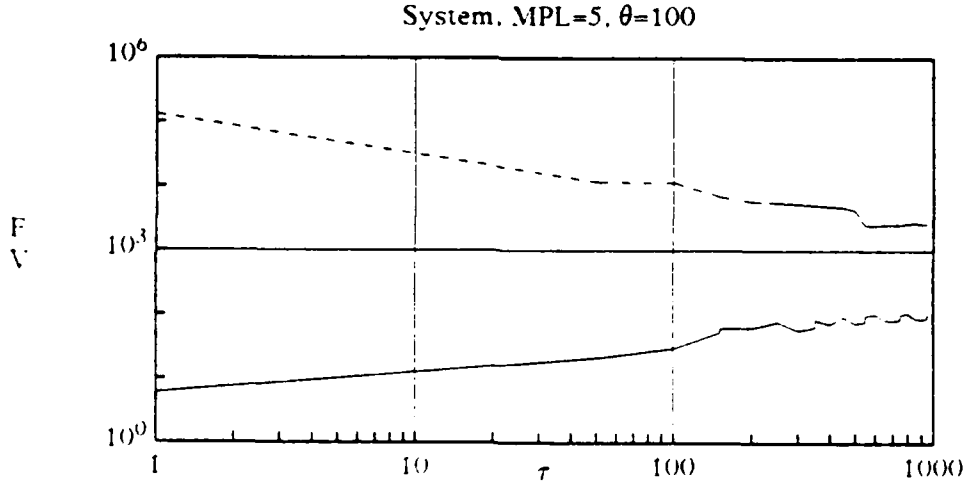


Figure 2-8: Page faults and average virtual memory versus τ

The fact that a τ-F anomaly does not necessarily accompany a τ-V anomaly, implies that WS may overestimate the size of a running process's working set, since a reduction in the working set size may not result in a subsequent increase in the fault rate; instead, the fault rate continues to decrease. In other words, WS may accumulate in the working set of a process more pages than it actually requires. This is especially true during interlocality transition periods. However, it is also possible for WS to accumulate redundant pages during the execution of a phase, rather than in transition between phases. Assume that a program contains a large locality structure (phase A) and several smaller phases. A properly tuned WS should be able to cover the locality comprised by phase A. The choice of a τ value, large enough to cover phase A, may result in covering several smaller phases before or after executing phase A. As a result of choosing large value for τ, some pages from previous phases may continue to be members of the working sets. Thus, the conclusion that "the WS serves as a dynamic measure of a program's memory demand" [20] is not accurate. The results reported in this section show that WS may overestimate the memory requirements of a program.

2.6.3. Virtual memory-fault rate anomalies

A virtual memory-fault rate anomaly (V-F) in a multiprogramming system exists for some θ, τ_1 and τ_2, if

$$V(\theta,\tau_1) > V(\theta,\tau_2) \text{ and } F(\theta,\tau_1) > F(\theta,\tau_2) .$$

The existence of virtual memory-fault rate anomalies is due to the existence of only one of either the parameter-fault rate anomaly or the parameter-virtual memory anomaly in the same τ region. The existence of both anomalies in the same τ region eliminates the possibility of exhibiting a virtual memory-fault rate anomaly. This observation is illustrated in the following three cases.

(1) $\tau_1 > \tau_2$, $V(\tau_1) > V(\tau_2)$ and $F(\tau_1) > F(\tau_2)$, τ-F and V-F anomalies

(2) $\tau_1 > \tau_2$, $V(\tau_1) < V(\tau_2)$ and $F(\tau_1) < F(\tau_2)$, τ-V and V-F anomalies

(3) $\tau_1 > \tau_2$, $V(\tau_1) < V(\tau_2)$ and $F(\tau_1) > F(\tau_2)$, τ-F and τ-V anomalies

In the first case, there exist a virtual memory-fault rate and a parameter-fault rate anomalies; however, there exists no parameter-virtual memory anomaly. In the second case, there exist a virtual memory-fault rate and parameter-virtual memory anomalies but not a parameter-fault rate anomaly. In the third case, both parameter-fault rate and parameter-virtual memory anomalies exist but the virtual memory-fault rate anomaly does not exist. All of these cases do in fact exist, as was shown in the previous section in Figures 2-7 and 2-8.

The virtual memory-fault rate anomalies are, graphically, illustrated in Figure 2-9 where we plot the page fault rate as a function of the average virtual memory for program INIT for θ = 30 and MPL=3. The anomalies in the figure are indicated by the increasing portions of the curve. V-F anomalies exist at the system level as well. In the previous subsection we observed that τ-V anomalies are not always accompanied with a τ-F anomaly, a condition necessary for the existence of V-F anomalies.

The V-F anomalies are particularly significant since they distort the shape of a life time curve, which is the inverse of the fault rate plotted versus the average virtual memory. Life time curves are used to model program behavior. Besides, some optimal multiprogramming management

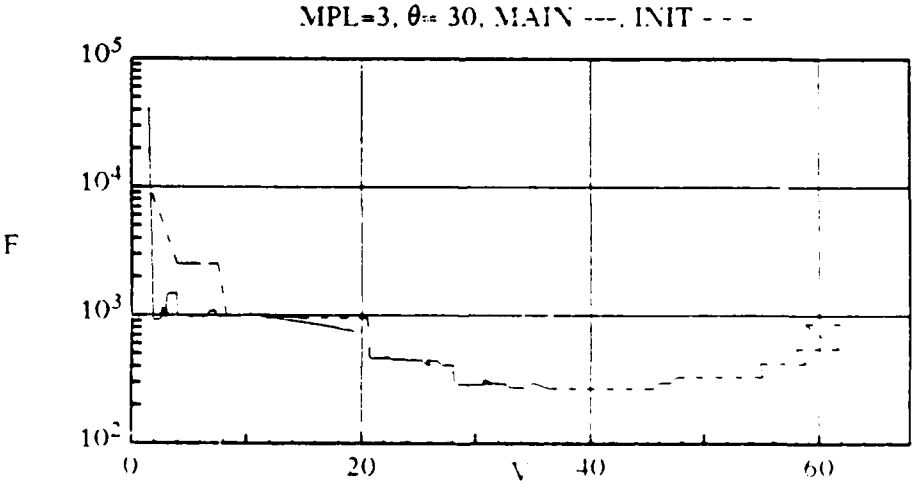


Figure 2-9: Fault rate virtual memory anomalies

strategies make use of life time curves, e.g., the primary knee criterion [20]. Most importantly, V-F anomalies prove that WS tends to accumulate more pages in the working set of a program than it actually needs. Furthermore, the existence of V-F anomalies suggests that the working set of a process need not be prepaged into main memory after it has been swapped out. In fact, swapping allows a process to re-evaluate its working set and demand paging, after a swapping operation, allows a process to remove redundant pages which could have accumulated in its working set.

2.6.4. System memory-fault rate and system memory-virtual memory anomalies

One would like to control the fault rate of individual processes or of the entire system by controlling the amount of memory available on the system. Such control is viable if the fault rate does not increase when θ increases. System memory-fault rate anomaly (θ-F) exists if, for some θ_1, θ_2 and τ

$$\theta_1 > \theta_2 \text{ and } F(\tau, \theta_1) > F(\tau, \theta_2)$$

where F is the fault rate of one process or of the entire system. This anomaly type can exist only in multiprogramming systems where the amount of memory available on the system dynamically changes. Increasing the maximum memory allowable on the system can be thought of as a means of reducing the page fault rate of individual programs or of the whole system. Contrary to one's expectation the fault rate may increase with increasing the maximum memory available on the system.

Our empirical results show that WS exhibits θ-F anomalies for both system fault rate and the individual processes' fault rate. For MPL=3, the fault rate achieved with θ=14 is larger than that achieved with θ=12 for τ values 20-30. For MPL=4, the fault rate achieved with θ=150 can be larger than that achieved with θ=100 by as much as 1512 faults, as shown in Table 2-5. Similar observations are made for MPL=5 and 10. For example, for τ=151 and θ_1=100,θ_2=150, $F(\tau, \theta_2) > F(\tau, \theta_2)$.

System memory-virtual memory anomalies (θ-V) exist in the same way as do system parameter-fault rate anomalies. θ-V exists if, for some θ_1, θ_2 and τ

Table 2-5
System memory-fault rate anomalies

τ	θ=100	θ=150	ΔF
15	8768	10280	1512
255	5123	5125	2
265	5054	5059	5
385	4403	4408	5
395	4334	4336	2

$$\theta_1 > \theta_2 \text{ and } V(\tau,\theta_1) < V(\tau,\theta_2)$$

i.e., the average virtual memory allocated to a process decreases instead of increases when θ is increased.

The anomalies reported in this section are not exclusive. There are many other anomalies, of all discussed types, which are not reported here; however, the figures and tables presented in this section are sufficient to illustrate the anomalous behavior of WS in multiprogramming systems.

2.6.5. Explaining the anomalies

The WS policy is a local dynamic memory management policy and, therefore, the programs in a multiprogramming system may affect each other's working sets through swapping as discussed earlier in Section 1 of this chapter. The swapping activity, thus, may be responsible for the unpredicted paging behavior. Our empirical results show that the swapping activity in a multiprogramming system is indeed the main reason for the existence of anomalies. To illustrate this observation we record for each τ the swapping rate, S (θ,τ). The swapping rates associated with parameter-fault rate anomalies of program INIT (MPL=3) are presented in Table 2-6a. This table includes all the anomalies exhibited by program INIT in order to illustrate the effect of swapping on the fault rate. Consider, for example, the table entry for θ = 11, τ_1=21, and τ_2=66. The fault rate increase, ΔF, is 262 page faults and $S(\theta,\tau_2) = 2635 > S(\theta,\tau_1) = 1600$. Note that the swapping rate increase, $S(\theta,\tau_2) - S(\theta,\tau_1) = 1635$, is much larger than the fault rate increase, $\Delta F = 262$. The reason for this difference is that not all the pages, previously swapped out, will have to be paged

Table 2-6a
Parameter-fault rate anomalies and swapping rates (INIT)

θ	τ_1	τ_2	$F(\theta,\tau_1)$	$F(\theta,\tau_2)$	ΔF	$S(\theta,\tau_1)$	$S(\theta,\tau_2)$
6	6	11	3406	4446	1040	1338	2710
6	126	131	1932	3695	1763	1738	3687
6	376	381	3686	3704	18	3678	3695
7	6	11	3108	3771	663	964	345
7	61	66	3166	3256	90	3027	3120
8	6	61	3038	3266	228	718	3140
8	126	131	3241	3248	07	3133	3146
9	6	16	2610	3122	512	300	2437
9	21	66	2818	2911	93	2396	2775
9	191	196	1537	3098	1561	1350	2929
10	6	16	2556	3189	633	138	2299
10	21	66	2753	2840	87	2237	2669
10	131	141	2823	2830	07	2678	2681
10	256	261	2781	2788	07	2632	2642
11	6	16	1525	3068	543	21	2178
11	21	66	2550	2812	262	1600	2635
11	71	131	2799	2813	13	2624	2637
11	196	206	1445	1455	10	1281	1287
11	321	381	1393	2836	1443	1343	2785
11	386	516	1395	2829	1434	1213	2796
12	6	21	2525	2558	30	14	666
12	36	41	1548	1587	39	1263	1374
12	201	205	1404	1418	14	1224	1241
12	256	326	1412	1493	81	1238	1311
12	381	391	1472	1513	41	1290	1331
12	761	766	1412	1424	12	1328	1341
13	26	31	1602	1670	68	634	790
13	36	41	1572	1785	213	998	1246
13	96	101	1505	1512	07	1287	1298
13	196	256	1365	1444	79	1185	1242
13	321	376	1404	1458	54	1222	1276
13	506	511	1308	1333	25	1222	1248
13	576	581	1306	1362	56	1223	1279
13	761	766	1326	1367	41	1243	1285
30	71	76	960	980	20	20	51
30	261	266	459	477	18	189	207
30	396	406	407	448	41	141	182
30	701	731	287	314	27	124	155
30	781	791	290	298	08	147	155
30	1101	1201	271	299	28	128	156
30	2201	5101	268	857	589	125	714
100	1101	1201	175	223	48	00	68

into memory in the future. The working set of a program, at the time of swapping, contains pages not related to the program's current locality. These pages have been resident in memory since they

were paged in; they remained in the working set of a program until a swapping operation occurred. Accumulation of unnecessary pages is viable because the window size, τ, can be large enough to cover more than one of the program localities, as has been discussed earlier. Similar to V-F anomalies, this observation suggests that the working sets of a swapped out program need not be brought entirely into memory once the program is rescheduled for execution. Such a strategy is further supported by the fact that a swapping rate increase does not necessarily produce fault rate increase. The choice of this strategy in this study is, therefore, justified. Moreover, it further weakens the claim that the WS serves as a measure of program demand.

The swapping activity is also responsible for parameter-virtual memory anomalies (see Table 2-6b.) This is obvious, since a swapping operation removes the working set of a process from main memory, thus equating the working set size to zero. This by itself does not generate anomalies. Anomalies by definition are related to the WS parameter τ. Therefore, if the swapping rate generated under a larger value of τ is more than that generated under a smaller value of τ, then there

Table 2-6b
Parameter-virtual memory anomalies (INIT)

θ	τ_1	τ_2	V (θ,τ_1)	V(θ,τ_2)	S(θ,τ_1)	S(θ,τ_2)
6	126	131	8.31	3.69	1736	3687
7	6	11	3.8	3.6	964	3455
7	16	21	3.85	3.76	3009	3024
8	6	11	3.84	3.71	718	2782
8	256	261	5.59	4.86	3158	3180
9	191	196	9.54	6.68	1350	2929
10	321	326	8.00	5.17	2639	2768
10	756	761	5.19	5.07	2752	2755
11	241	246	10.14	8.59	1304	1369
11	271	376	12.22	5.72	1220	2794
11	386	391	12.38	5.65	1213	2795
13	321	326	11.89	11.84	1221	1231
30	766	771	31.37	30.62	138	147
30	3401	3501	55.12	54.81	187	280
30	5001	5101	61.83	58.68	404	714
100	1101	1201	37.97	37.90	00	68

is a chance for the anomalies to appear. A plot of the swapping rate of the system versus τ is given in Figure 2-10 for MPL=5 (θ=100, 150, 200). Figure 2-10 shows that the swapping rate is an increasing function of τ most of the time. Moreover, swapping occurs at relatively large values of τ. For θ=200, the swapping rate curve is shifted to the right of that for θ=150 and θ=100; swapping occurs at larger values of τ.

A swapping rate increase that results in a parameter-virtual memory anomaly, but not in a parameter-fault rate anomaly, produces a virtual memory-fault rate anomaly as discussed earlier in this section. Hence, a swapping rate increase that results in a parameter-fault rate anomaly, but not in a parameter-virtual memory anomaly, results in a virtual memory-fault rate anomaly. Moreover, a system memory-fault rate anomaly has been shown to be preceded by a parameter-fault rate anomaly. Therefore, it can be concluded that the swapping activity in multiprogramming systems is the main reason for the anomalous behavior discussed in this chapter.

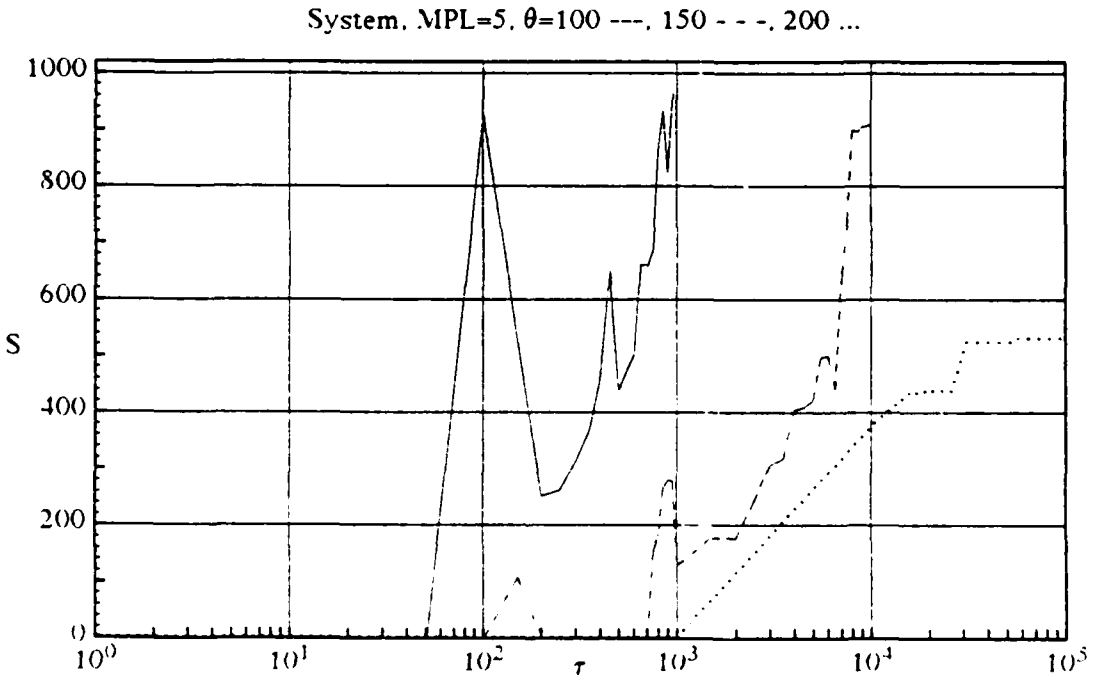


Figure 2-10: Swapping rate versus τ

2.7. Summary and Conclusions

This chapter has demonstrated WS anomalies in multiprogramming systems. The presence of anomalies is of theoretical interest in itself. However, we found that the anomalies are far too numerous to be considered only of pathological or contrived nature. Practically, the existence of anomalies complicate the control process of WS policy. The WS parameter, τ, may not be used in a straightforward manner to control the fault rate in the system and memory allocation. Moreover, WS anomalies, especially the parameter-virtual memory anomaly, illustrate how WS overestimate a process's working set and, hence, memory could be overcommitted during the execution of a process.

Furthermore, this study suggests that results obtained from uniprogramming studies should not be used in a simplistic manner to arrive at multiprogramming paging strategies. The WS policy exhibits only certain types of anomalies in a uniprogramming system. In a multiprogramming system, performance measures depend not only on the intrinsic behavior of a program but also depend on the behavior of other processes in the system. Interaction between processes takes place through paging in global policies such as global LRU and through swapping in local policies such as WS.

The WS anomalies, together with the WS high cost of implementation, leaves open the search for a better policy for managing memory hierarchies in multiprogramming systems. The next chapter presents a new approach to the memory management problem. A parameterless policy is proposed which can respond to the memory requirements of a program taking into consideration the requirements of other processes in the system.

CHAPTER 3

CD: A COMPILER DIRECTED MEMORY MANAGEMENT POLICY

The idea of using *memory directives* (MD) for the management of memory hierarchies in a multiprogramming virtual memory system (VM) has been hinted at by many authors. Madison and Batson [30] suggested that if program localities generated by the BLI model could be correlated to the source level code, then it would be possible for the compiler to generate MD to identify program localities at run time. Abu-sufah [5] suggested the use of data dependence graphs to isolate the localities at the source level. In his Ph.D thesis Abusufah found that the localities of numerical programs in a paged system generated by the BLI model are due to loop structures. A similar conclusion was made by Malkawi [31] for segmented systems. The use of memory directives for optimal memory management was also suggested by Hagmann and Fabry [27] and by Kearns and DeFazio [29]. Except for [5] and [31] none of the researchers have proposed any particular MD to be used. Abu-sufah proposed a directive called *allocate* which has the function of locking a page in memory if it can be identified as a member of a program locality. When the program moves to another locality phase, a *deallocate* routine is called to release those pages allocated during the execution of the previous locality. Abu-sufah suggested that a program has to be transformed [2] before allocate and deallocate can be effectively used. Program transformation requires the use of data dependence graphs to resolve data dependencies. The directives suggested by Abusufah fail to reflect the hierarchical structure of program localities which is a common locality characteristic [30]. Besides, allocate and deallocate can not respond to the dynamic change in the memory status of a multiprogramming system.

The idea of using MD has been practically implemented in real systems. Both VAX/VMS and Berkeley UNIX allow the user to lock and unlock some pages in physical memory. The effectiveness of such facilities in VAX/VMS was illustrated by Abaza [1], who showed that the performance of

some numerical algorithms can be enhanced if the directives provided by VAX/VMS are properly used. However, one would like to free the user from having to call a system routine to lock or to release a page, and having to isolate a page that should be locked in memory in order to achieve a better performance. Besides, a user may not be able to determine which page should be locked and when it should be released, unless he has the proper knowledge of his program behavior as well as the knowledge of the system.

In this thesis, three memory directives are designed to achieve two goals. The first one is to allocate X physical page frames to a running process's resident set. A directive, designed for this purpose, should be able to define the size of a program's resident set and allocate enough physical pages to accommodate it. In this study, such a directive is called *ALLOCATE*. The second goal is to lock a page or set of pages in main memory. A locked page, by definition, is exempted from being paged out by the page replacement mechanism. A directive is developed for this purpose, and called in this thesis *LOCK*. LOCK has a similar function to the directive proposed by Abusufah [5] and to the system facilities provided by VAX/VMS and Berkeley UNIX (VMS and UNIX user manuals). A page that has been locked in memory by LOCK is unlocked by a directive called *UNLOCK*. Later in this chapter, we shall discuss a case in which the operating system (OS) is entitled to release a page before UNLOCK does so. ALLOCATE, LOCK, and UNLOCK are discussed in greater detail in the following sections.

Based on the three directives developed in this study, a compiler directed memory management policy (CD) is proposed. CD operates as follows. At compile time, a preprocessor generates directives of the type ALLOCATE, LOCK, and UNLOCK. These directives are inserted at appropriate locations into the compiled object code of a user's program. At execution time, the directives are executed by the CPU. When a directive is executed, CPU generates a call to a particular OS routine responsible for processing and handling memory directives. Figure 3-1 presents a block diagram of CD.

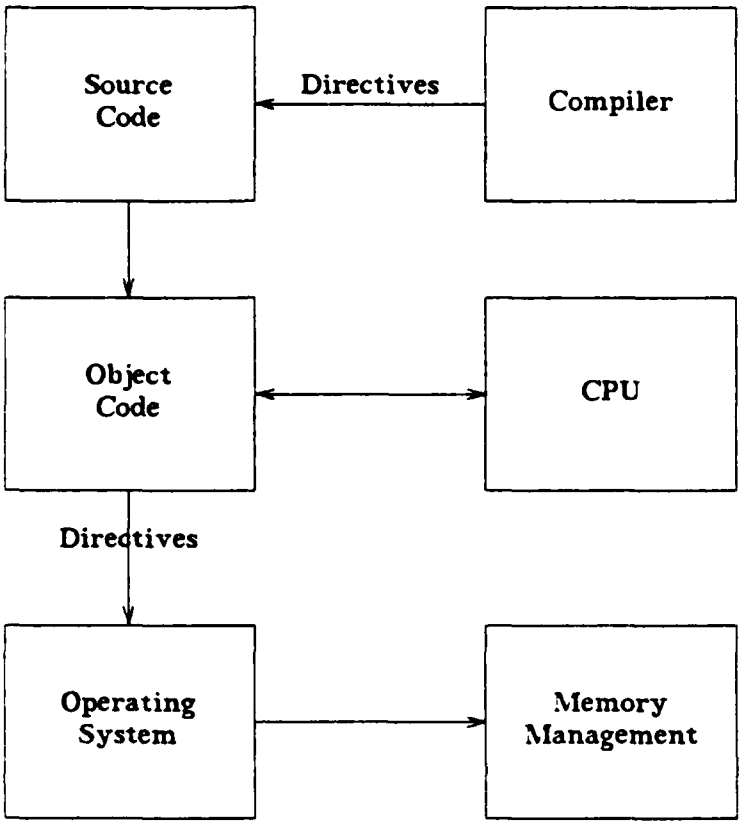


Figure 3-1: Block diagram of a compiler directed memory management policy

3.1. Memory Directive: ALLOCATE

One of the major problems a memory management policy has to solve is the amount of physical memory that should be allocated to a program during its execution. Run time policies, whether static or dynamic, determine the number of pages to be allocated at run time as discussed in the first chapter. It has been shown in Chapter 2 that WS, a dynamic run time policy, may overestimate a program's memory requirements. Compiler directed memory management policies estimate the memory requirements of a program at compile time, using source level information, and passed to the OS through ALLOCATE, which is designed in accordance with locality characteristics of program behavior and the constantly changing free memory space available on a multiprogramming VM system.

3.1.1. Locality characteristics of numerical programs

A locality structure may result from data structures created at run time, e.g. stacks, or from data structures declared in the source code of a program. The later case is considered in this thesis. The BLI model of program localities [30] suggests that array references inside loop structures of numerical programs are the main reason for the existence of localities at execution time [5], [31]. A nested loop structure produces a hierarchical locality structure. Such structure defines one of the locality characteristics, namely the level of a locality in the hierarchy of localities. Another characteristic of major significance to our study is the virtual size of a locality. The time duration is also a locality characteristic as seen in [5], [30], [31]. Consider Example 3-1 for illustration.

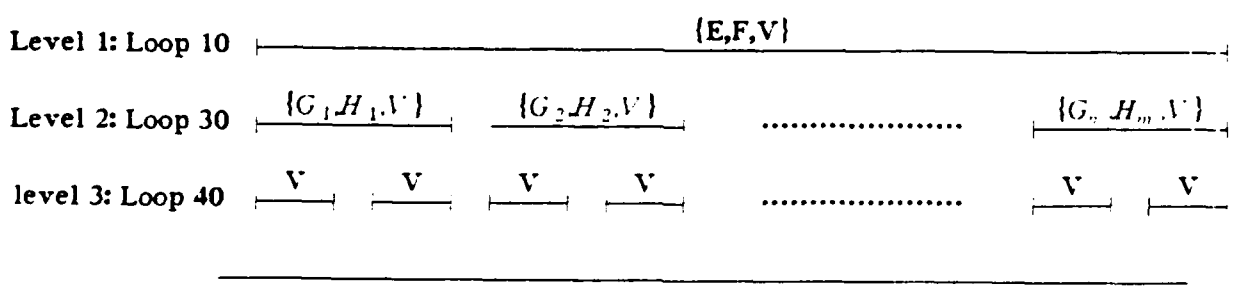
Example 3-1 shows a FORTRAN-ike piece of code. The maximum nest depth of the loop structure is three. Two arrays, E and F, are referenced inside loop 20. Arrays E and F are referenced in a row major order, i.e., the elements of a row are referenced while the current column index, I, is fixed. The elements of an array are stored in a column major order; this assumption

Example 3-1:

```
DO 10 I= 1, N
    DO 20 J=1, M
       E(I,J) = F(I,J)
    20   CONTINUE

    DO 30 K=1,M
       G(K,I) = H(K,I)
       DO 40 L=1,NN
          V(L) = V(L)*2
       40   CONTINUE
    30   CONTINUE
10   CONTINUE
```

The localities of the above code are illustrated in the following diagram:



holds throughout this thesis. Every element of arrays E and F is referenced one time during one iteration of loop 10, i.e., the entire virtual space of arrays E and F is spanned during one iteration of loop 10 due to a full execution of loop 20. Hagmann and Fabry called this type of referencing pattern *total* [27]. Note that there are M iterations of loop 20 per each iteration of loop 10. Therefore, a locality comprised by loop 10 includes the virtual spaces of E and F.

Loop 10 is the outermost loop which forms the highest level locality, or level one locality as termed in [30].

Arrays G and H are referenced in a column major order inside loop 30. When loop 30 executes, the column elements of arrays G and H are referenced sequentially, while the column index, I, is fixed at the outer loop level (loop 10). Since the elements of one column are stored in consecutive pages, according to the storage scheme, the locality at this level includes only the virtual space of the column being referenced in the virtual space of arrays G and H. The index I takes a new value only when loop 10 reiterates. The elements of a new column will be referenced during the next iteration of loop 30. In other words, the virtual space spanned during the execution of loop 30 is determined by the new column elements of G and H. Consequently, references to G and H inside loop 30 form a locality as long as loop 30 remains active. However, every time loop 30 resumes execution a new set of pages form the locality. In the diagram of Example 3-1, localities formed by loop 30 are illustrated by $G_1 H_1, \cdots, G_n H_n$, where G_i is the virtual size of column i of array G.

A one-dimensional array, vector V, is referenced inside loop 40. During the execution of loop 40, the virtual space of V is spanned totally. The virtual space of V is referenced totally during each iteration of loop 30 and loop 10. Therefore, V participates in the localities formed at level 1 and 2 as well as at level 3. The localities are illustrated, graphically, in the diagram of Example 3-1. Example 3-1 is too simple to illustrate how program localities can be automatically extracted from the source level code.

Our concern here is with the hierarchical characteristic of program localities. A *macroscopic* view of the locality structure exhibited in Example 3-1 shows that all arrays, E, F, G, H and V,

should be considered part of the program's current locality. This view is obtained by looking at loop 10 as indivisible entity. If the program's memory reference pattern is observed while the program is executing loop 40, the program's current locality appears to include vector V only. Such a *microscopic* view of the locality structure shows that the smallest program locality dominates all other localities. This illustrates how a program may change localities within a given locality structure, *(intra-locality transitions)*. In Example 3-1, intra-locality transitions occur between levels 1 and 2 and between 2 and 3.

The problem of intra-locality transitions was treated in [9] by linearizing the locality structure. To linearize a locality structure consisting of two levels is to decide that one of the locality levels is more significant than the other at some time instance. Least significant localities are dropped from the locality structure, thus leaving only one path connecting all locality levels. The difficulties of this approach are cited in [5] and [31]. Besides, each locality level in a locality structure reflects the memory referencing behavior of a program during a particular phase of the program execution. In Example 3-1, while the program executes loop 40, the virtual space of vector V is being referenced continuously, irrespective of the significance of level 3 locality compared to level 2 or 1. Therefore, the locality comprised by loop 40 is significant during the execution of loop 40 and the locality comprised by loop 30 is significant during the execution of loop 30 and so on. Allocating the outermost loop produces the minimum possible fault rate for a given locality structure, irrespective of its relative significance to other levels, since the virtual spaces of all referenced arrays within the outer loop are made resident in memory. However, it may not always be possible to allocate the locality comprised by the outer most loop (level one locality) due to insufficient free memory. In such cases, the availability of free memory should determine which level of the locality structure should be allocated.

From the above discussion the following observations are made. The highest level locality (level 1) produces the lowest possible fault rate, when allocated completely in main memory. That is because every page referenced inside a level one locality is paged only once into main memory

(assuming a demand paging strategy), and will not be replaced by the page replacement strategy. The allocation of a level one locality implies that the resident set of a program should not be less than the number of pages referenced inside the locality. However, if a level one locality is too large to fit in the main memory, the next lower level locality should be considered for allocation (lower level localities have a smaller size than higher level localities). In other words, a program settles down to a microscopic view of its locality structure. If the second level locality can not be allocated, the third level locality is tried for allocation, and so forth. A program may keep reconsidering its lower level localities for allocation as long as there exists at least one more lower level locality. The program should not, however, be allowed to run if the lowest level locality can not be allocated; this restriction is necessary to prevent thrashing. Assume that the lowest level locality contains N pages and there are only N-1 free memory pages. N-1 pages from the lowest level locality may reside in main memory and one page has to be maintained in virtual memory. Every time a reference is made to the N^{th} page (in virtual memory), a page has to be removed from the main memory. A reference to a replaced page, in the future, will cause a page fault which results in replacing another page. The outcome of this cyclic faulting process is a short life time between successive faults, a phenomenon known as thrashing.

These observations lead to two key principles underlining the design of the ALLOCATE directive. First, the highest level locality, level one in the hierarchical locality structure, is favored over localities of lower levels for allocation purposes. Secondly, the lowest level locality in a hierarchical locality structure, imposes a lower limit on the memory space that should be allocated to a running process. These two principles reflect the dynamic change of the program's memory demand due to intrinsic properties of program behavior. The failure to recognize these principles may lead to improper memory allocation strategies. In order to incorporate the above principles into MD, each locality at some level in the hierarchical locality structure is assigned a priority index, P.

Up to this end one can recognize two primitives for ALLOCATE. The first one is the amount of memory to be allocated, X, given by the virtual size of a locality. The second one is the priority

of allocation, P. ALLOCATE may have the following form

$$ALLOCATE(P,X)$$

where X is the virtual size of a locality, and P is the priority index associated with that locality. Upon executing a directive of type ALLOCATE, a request is issued to the operating system to allocate X pages, given that the priority of allocation is determined by P. Both primitives, P and X will be discussed in more detail in Section 3.1.4.

ALLOCATE, in its simple form given above, can not respond to the dynamically changing amount of free memory space in a multiprogramming system. The amount of free memory space available on the system may increase if a running process completes its execution and returns to the system, whatever memory it has occupied, or if a process enters a new phase with a smaller size locality, thus, adding the released pages to the free memory. On the other hand, the free memory may shrink in size if a new process is added to the system or if a running process enters a new phase with a larger size locality. Moreover, the above form of ALLOCATE does not completely incorporate the first principle cited above: namely, that higher level localities should be favored over lower level ones. To account for these two drawbacks, a more complex form of ALLOCATE directive is given below:

$$ALLOCATE(P_1,X_1)\ else\ (P_2,X_2)\ else\ \cdots\ else\ (P_n,X_n)\ \text{where}\ X_1 \geq X_2 \geq \cdots \geq X_n$$

Each ALLOCATE directive has one or more parameters. Each parameter has two primitives enclosed in parentheses "(P,X)". At any level of a locality hierarchical structure, ALLOCATE contains a parameter associated with the current level and one parameter for each level enclosing the current level. The order of parameters in ALLOCATE is such that parameters associated with higher level localities precede those associated with lower ones, as shown in Figure 3-2. A multinested loop structure is shown in Figure 3-2. Each loop forms a locality with size X_i and has a priority P_i. The outermost loop forms a level one locality with X_1 and P_1. The directive associated with this locality is ALLOCATE (P_1,X_1). Going down in the hierarchy structure to the second loop with nest depth 2, the directive reconsiders the allocation of the previous locality specified by (P_1,X_1) before it considers the primitives of the second level locality specified by (P_2,X_2), and so

forth.

3.1.2. Processing of ALLOCATE directive by the operating system

For the moment, we assume that directives of the form

$$ALLOCATE\,(P_1,X_1)\ else\ \ (P_2,X_2)\ else\ \cdots$$

have been inserted into the program's code at compile time. At run time the directives are executed by the CPU. Once a directive is executed, a system routine is invoked to handle its processing. ALLOCATE issues requests of the form $(P_1,X_1),(P_2,X_2),\ldots$ in the same order. The OS first receives

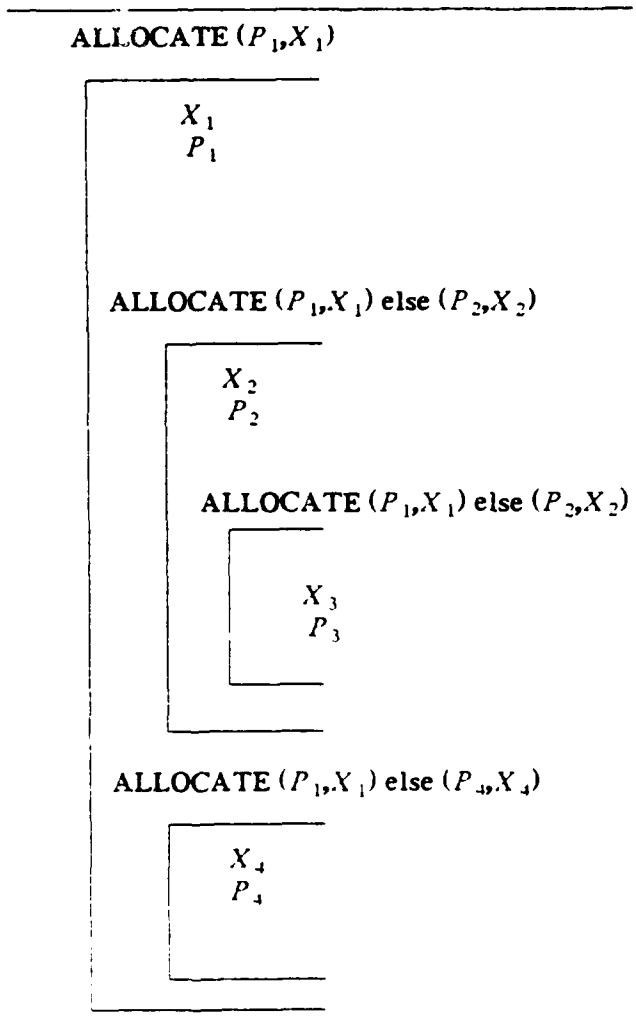


Figure 3-2: Example of ALLOCATE directive

the request (P_1,X_1) and tries to allocate X_1 pages from the available free memory. If X_1 pages can not be allocated, then the OS examines the value of P_1. As a convention P=1 is chosen to be the priority of the lowest level locality in a hierarchical locality structure. Hence, $P_1>1$ means that there is at least one more lower level locality, and at least one more directive argument (P_2,X_2) where $X_2<X_1$ and $P_2<P_1$. In this case, the program is allowed to continue its execution, with its current memory allocation from the previous directive, until the next request (P_2,X_2) is received. Once again, if X_2 can not be allocated, the execution continues only if $P_2>1$. This process continues until a memory request X_i is allocated, or the priority of the request is $P_i=1$. In other words, the program exists in the scope of its lowest level locality, or, using source level code notation, the program is currently executing the innermost loop of a multi-nested loop structure. In this case, OS either suspends the program's execution or invokes a swapping mechanism (SM). The choice between these two actions depends on the priority of the running job and the priorities of other jobs existing in the system at the time of processing a directive. In the performance evaluation of CD it was assumed that all processes have the same priority and, thus, the OS invokes SM whenever it has to make a choice. SM is discussed below in greater details. The processing of ALLOCATE is shown in Figure 3-3.

In Figure 3-3 the priority index P=1 is used to indicate the lowest level locality. With P=1 associated with the lowest level locality, the OS simply checks whether the current priority is larger than one or not in order to determine the next step when sufficient memory can not be allocated. Otherwise, if P=1 is associated with the highest level locality and P is increased with the increase of the depth of the locality structure, a look-ahead scheme will be necessary to know the relative position of the current locality. However, assigning P=1 to the lowest level locality, comprised by the innermost loops, inhibits the use of a one pass top down parsing scheme when the directives are inserted, as will be seen in Section 3.1.4.

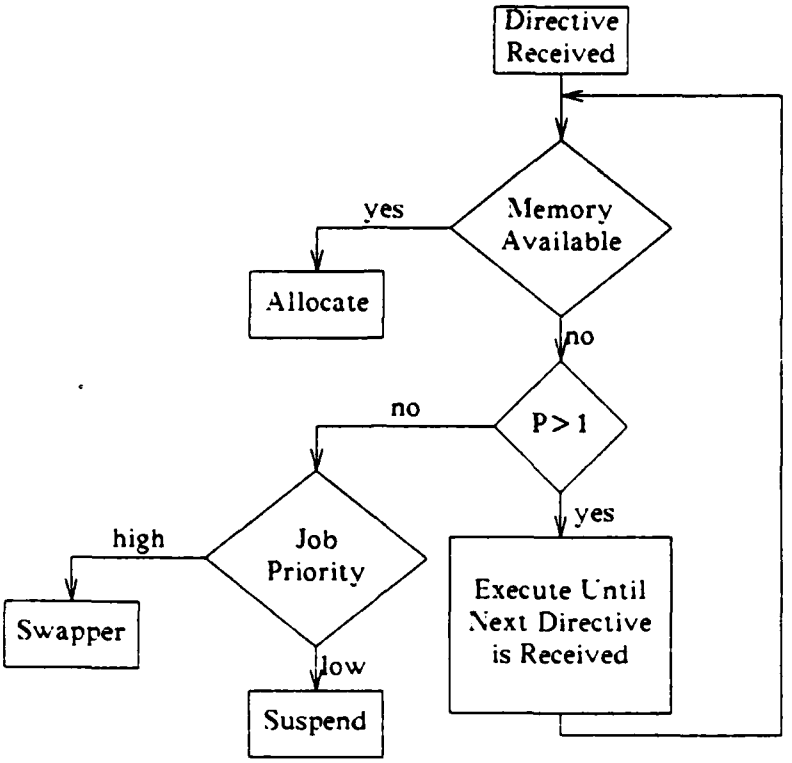


Figure 3-3: Directive processing by the OS

3.1.3. Swapping mechanism

The OS may invoke a swapping mechanism (SM) if the available memory space is not enough to allocate the current request and the priority of the current request is P=1. Besides being able to invoke SM, CD provides a strategy for partial swapping, using the priority primitive of ALLOCATE.

In regular swapping strategies, a process is selected for swapping, according to some criteria, and its resident set is removed from main memory. We call this strategy *total swapping* as opposed to *partial swapping* strategy (PS). Partial swapping reduces the current resident set of a process, selected for swapping, to a smaller value. The viability of partial swapping is facilitated by the priority primitive and the hierarchical nature of ALLOCATE. PS operates as follows. When invoked by a directive with P=1, the swapper searches for any process occupying memory space X,

with a priority $P_i > 1$. The resident set of such a process is reduced from X_i to a new value X_j. X_j is the size of a lower level locality with $X_j < X_i$ and $P_j < P_i$. In the model used in our study we reduce the resident set size of a process to that one associated with P=1. The philosophy behind partial swapping is that a process A may find enough memory space to allocate its largest locality, while process B can not allocate its smallest locality. This may happen if process A is scheduled to run when the system is not heavily loaded, while process B enters the system when it is heavily loaded. Forcing all processes to run with their smallest localities allows more processes to share memory. However, thrashing is prevented by ensuring that every process is allocated enough memory to accommodate one of its localities, no matter how small the locality is.

Various schemes of partial swapping could be implemented. For example, a multiple queue could be used to hold processes with different directive priorities. The partial swapping mechanism would transfer the processes in the largest priority queue to the next lower level and continues until either the memory request is satisfied or the only unempty queue is the one with P=1. Total swapping becomes necessary if every process in the system is running with P=1. Partial swapping strategy is further illustrated in Example 3-2.

Example 3-2:

Assume that two processes A and B are running in a system with 120 memory pages. A executes the directive MD_A: *ALLOCATE* (3,100) *else* (2,50) *else* (1,10) and B executes MD_B: *ALLOCATE* (1,25). Assume further that A is activated first. The following execution time intervals (t_i) are observed:

t1: A executes MD_A. The first request (1,100) is granted since X=100 is less than the available free memory F=M-0=120. The status of A is S_A=100, P_A=3 (S is the resident set size); the last argument of MD_A (1,10) is saved in a process specific record. F=120-100 =20 pages.

t2: Interrupt occurs and B is activated.

t3: B executes MD_B. The first request (P=1,X=25) can not be granted because X=25 is larger than F=20. Since P=1, OS invokes SM. The partial swapper (PS) finds A occupying 100 pages with a priority larger than one, P=3. PS reduces S_A from 100 to 10 pages; S_A: (P=3,X=100) → (P=1,X=10). F= 120-10 = 110 pages. Now B's request can be granted; S_B=25 pages. F=110-25=85 pages.

t4: Interrupt occurs and B is activated.

t5: A executes MD_A. The first request (P=3,X=100) can not be granted because X=100 > F=95. A continues execution with its previous allocation S_A=10 until the next request (2,50) is received. The request is granted since X=50 < F=85. The status of A is S_A = 50 and P_A=2, F=45.

A steady state is reached with 25 pages allocated to B and 50 pages allocated to A. B always gets

the 25 pages it asks for since the request has a high priority P=1. A cannot be allocated 100 pages as long as B is in the system; A will not, however, be forced to run with 10 pages since B cannot invoke SM anymore.

3.4.1. Primitives of ALLOCATE directive

ALLOCATE incorporates two primitives: the priority index P and the memory request X. Both primitives are discussed in the following subsections.

3.4.1.1. Priority primitive, P

The hierarchical nature of memory demands due to the hierarchical nature of locality structures is reflected into the hierarchical form of ALLOCATE through the *priority primitive* P. Recall that the allocation of the highest level locality, level one, achieves minimum page fault rate during the execution of a multi-nested loop while the allocation of the lowest level locality, comprised by the inner-most loop, is sufficient to prevent thrashing. The highest level memory demand is determined by the highest level locality, comprised by the outermost loop of a multi-nested loop structure, whereas lowest level memory demand is given by the virtual size of the lowest level locality. Memory requirements in between the outermost and innermost loop of a program are defined by the sizes of corresponding localities.

The priority primitive, P, is used to determine the sequence in which localities of a given construct should be tried for allocation. A locality at level one should be tried for allocation before a locality at level two, and a locality at level two should be tried for allocation before a locality at level three, and so forth. Such precedence is motivated by the hierarchy of locality sizes. Higher level localities are larger in size than lower ones. And the allocation of larger localities is sought to achieve lower fault rates.

The priority primitive is used to impose a lower limit on the memory requirement of a program. Such a lower limit is given by the virtual size of the lowest level locality. The inability to allocate the lowest level locality results in thrashing. The execution of a directive associated with a lowest level locality requires the allocation of such locality even at the expense of swapping

some processes out of memory. Thus, P is used by OS to invoke SM when necessary. Moreover, P is used by the partial swapping mechanism as discussed in the previous section. A process, running with P>1, might be selected for swapping by PS.

The value of P can be deduced from the relative position of a locality in a hierarchical locality structure. The largest value of P is defined by the maximum nest depth (Δ) of a loop structure since Δ imposes an upper bound on the number of localities in a given locality structure. Hence, the values of P range from 1 to Δ. The outermost and the innermost loop compose an envelop enclosing all other intermediate localities. P=1 can, in principle, be assigned to either one and P=Δ to the other. In the previous section we assigned, by convention, P=1 to the innermost loop. The motive behind this is to enable OS, while processing a directive, to determine the memory request associated with the lowest level locality. This is necessary for two reasons. First, if the current memory request can not be allocated and P=1, SM should be invoked. Otherwise (if P=Δ is assigned to the lowest level locality), Δ should be known at the time of executing a directive to compare it with P every time a request can not be satisfied. The second reason, the value of the (P,X) pair associated with the lowest level locality should be stored in order to partially swap a process if needed.

In a multi-nested loop structure, there can be more than one innermost loop. Each of these loops forms a lowest level locality which must be allocated if the process is to continue execution. The priority P=1 is assigned to every innermost loop and P=Δ to the outermost loop. The priority of any intermediate level takes the value between 2 and Δ-1. Such value is used to indicate how many more parameters a directive could have. In effect the value of P at any level L_i is a measure of the distance d between L_i and the innermost loop enclosed by L_i. The priority P_i assigned to any loop L_i can be iteratively evaluated by finding the maximum nest depth Δ_i of an inner loop enclosed by L_i, assuming that L_i is an outermost loop, and assigning $P_i = \Delta_i$.

Assigning priorities to a loop structure, thus, cannot be performed with a single top down parsing technique since it is necessary to know the depth of the innermost loop relative to the

current outer one. A single top down parsing scheme can be used, however, if P=1 is assigned to the outermost loop. A procedure for assigning priorities to various loops in a multi-nested loop construct is given in Algorithm 3-1.

Algorithm 3-1: Assign priorities to loop structures;

Repeat:

Step 1: Parse until a loop is encountered.

Step 2: Find the maximum nest depth, Δ, related to this loop.

Step 3: Assign P=Δ to the current loop.

Until the end of the program is reached.

An example using Algorithm 3-1 is shown in Figure 3-4.

3.1.4.2. Memory request primitive, X

The memory requirement of a program, X, at a given time, is determined by the virtual size of the current program locality under execution and is used as a primitive of ALLOCATE. In this study the localities are restricted to those comprised by loop structures since the study is conducted on numerical programs where the locality structures can be correlated to loop structures at the source level code [5], [30] and [31]. In this section, the virtual size of a program locality is estimated using source level information.

Only references to array data structures are considered in this study. The instructions code and data constants are assumed to be locked permanently in main memory. This assumption is realistic since the paging behavior of numerical programs is dominated by references to array data structures inside loops [4], [30]. Moreover, the virtual size of the instructions and the constants is relatively small compared to the virtual size of array data structures.

The estimation of the virtual size of the current locality utilizes only the information available at the source level code. A wide range of FORTRAN programs used in different packages was examined for the purpose of identifying their localities, using the information inherent in

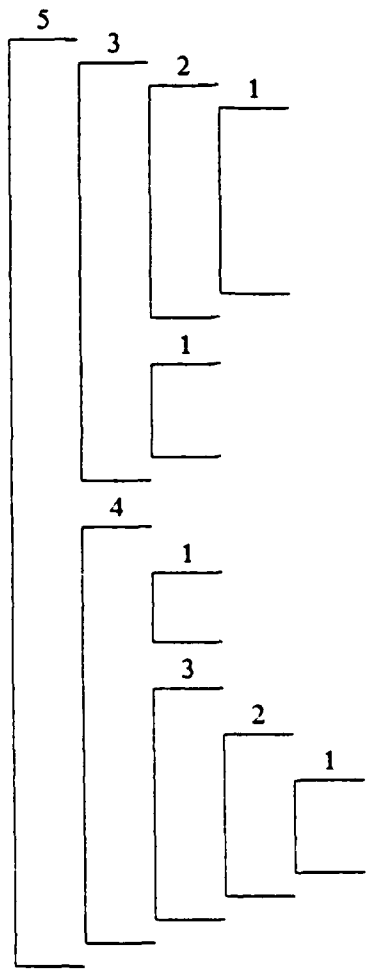


Figure 3-4: Example of assigning priorities

the source code. Some of these packages are UIARL : University of Illinois Atmospheric Research Lab. ACM: ACM Standard Programs. IEEE: IEEE Standard Programs for signal processing; NRL: Naval Research Laboratory. AFWL: Air Force Weapons Laboratory. Fishpak, Eispak, Minpak. Fishpak is a package of Fortran subprograms for the solution of separable elliptic partial differential equations developed at NCAR (National Center for Atmospheric Research). Eispak is a package of Fortran subroutines for the analysis of standard and generalized eigenvalue programs. Minpak is a package of Fortran subroutines for finding the minimum of solution squares of sets of nonlinear equations.

Examining the source code of these programs reveals that six parameters can be used to calculate the virtual size of a locality. Five of these parameters are program dependent and one is system dependent. The system dependent parameter is the page size (P). The page size is necessary for calculating the virtual size of a locality in pages, since memory allocation is measured in pages. Program dependent parameters are

(1) Array size (Σ) : Σ is usually given as (M x N) dimension, where M is the number of rows and N is the number of columns. A vector is an array with N = 1. Only up to two-dimensional arrays are considered in this study. Array sizes are given explicitly in dimension declaration statements. The virtual size of an array (S_A) is given by

$$S_A = \frac{(M \times N)}{P}$$

assuming that each array element is one word long. The virtual size of all arrays referenced in a program comprise an upper bound on its memory requirements. The memory requirements during the execution of a loop structure are bounded by the virtual size of the arrays referenced inside the structure.

(2) The nest depth of a loop structure (Δ): Δ determines whether the current locality has a hierarchical structure or not. The value $\Delta > 1$ implies that a hierarchical locality structure with utmost Δ levels may exist. It is possible, however, not to have a hierarchical locality structure with $\Delta > 1$. For example, a doubly nested loop ($\Delta = 2$) with arrays referenced in a row major order inside the inner loop forms a single locality of level one. The nest depth is also useful for assigning priority indexes to nested loops.

(3) The number of indexed variables used to reference the elements of an array (N): N is used to give an upper bound on the number of distinct array pages referenced at a given locality level. The maximum number of array elements which can be referenced during one iteration of a loop is determined by the number of distinct indexes, N, used to address the array. If the array elements referenced at a particular level are stored in distinct pages, then N distinct pages are referenced at this level. Depending on the dimension and the order of reference of

an array, N can be used to give an upper bound on the number of array pages which participate in the formation of the locality at the current level.

(4) The order in which arrays are referenced has a direct effect on the formation of a locality. If an array is referenced in the same order as the elements are stored in the virtual storage, then each array referenced inside a loop contributes to the locality comprised by that loop. On the other hand, if the elements of an array are referenced in a different way than that of the storage scheme, then references fall across pages. In this study we have assumed a column major order scheme. The elements of the first column are stored sequentially in the same page. If the number of elements in a column exceeds the number of words in a page, a second page is used, and so on until all elements of a column are stored. Then the elements of the second column of the array are stored in the same manner, until all columns have been stored.

An array is said to be referenced in a column major order, A^c, if the column index, J, is fixed and the row index, I, varies during the execution of a loop, L_i. The addresses generated by references to A^c fall into adjacent virtual space locations. Once a page is addressed, its elements will be referenced sequentially until a second page is addressed. When a second page is referenced, the first page will no longer be active. Therefore, only one page from A^c will be active at a time. However, if several row indexes are used to reference a column's elements, several pages might become active during the execution of L_i. Hence, the locality comprised by L_i may consist of several pages, depending on the number of row indexes used in combination with a column index. It is also possible that several column indexes, J, could be specified at outer levels. In this case, the virtual space specified by each column will participate in the formation of CL.

An array is referenced in a row major order, A^r, if the elements of a row are referenced sequentially during the execution of a loop, L_i. The row index I is fixed, while J, the column index, varies inside L_i. Any elements referenced in a row major order are located in two different columns and, hence, in two different pages, unless the column size is less than the page size. Therefore, the number of pages to be referenced during the execution of L_i is

equal to the range of the column index J. A page referenced in one iteration of L_i would not be referenced in the next iteration, since the next reference is made to an element in a different column. Hence, no locality of reference exists at L_i level. If the virtual size of a column, S_c, is less than the page size, the elements of two successive columns may be stored in the same page; assuming that all pages of an array are filled except, possibly, for the last page. In this case the number of pages expected to be referenced during the execution of L_i is equal to the total number of pages in the virtual space of A'. The maximum number of pages referenced from the virtual space of A' depends on the size of each column compared to the page size. Given a A' with a dimension (MxN), where M is the number of elements in a column or the range of the row index I, and N is the number of elements in a row or the range of the column index J, the following equation finds the maximum number of pages, X_i, that may be referenced during the execution of L_i:

$$X_i = \begin{cases} \dfrac{M*N}{P} & \text{if } S_M < P \\ N & \text{if } S_M \geq P \end{cases} \qquad (3\text{-}1)$$

where S_M is the virtual size of a column and M*N/P is the virtual size of the array. Note that references to A' do not form a locality of reference at the same level of reference (L_i). However, a macroscopic view from a higher level L_j, where j=1.....i-1 shows that a locality of reference results from references to A' at L_i.

(5) The level (or nest depth) at which an array is referenced (λ): λ = 1 is the nest depth of the outermost loop in a multi-nested loop structure. λ increases as we go deeper into the loop nest. The nest depth of the inner most loop, $\lambda = \Delta$, is the maximum nest depth of a loop structure. The smaller the value of λ, the higher is the level. A row-wise referenced array at some level $\lambda=i$ does not form a locality at this level. However, if there exists a higher level $\lambda<i$, then A' forms a locality at all levels with $\lambda<i$. That is because the virtual space from A', referenced at $\lambda=i$, is rereferenced repeatedly during the execution of any higher level loop with $\lambda<i$. The entire virtual space of A', $S_{A'}$, is referenced during each iteration of any loop with

level $\lambda=1.2...i-2$. Therefore, A' tends to form a locality at higher levels $\lambda<i$ with a size X_{i-1} given by equation 3-1 for the locality at level $\lambda=i-1$ and a size $X_j=S_{A'}$ for $j=1.2,\cdots,i-2$.

Similarly, for the case of a vector, one iteration of a higher level loop $\lambda=j$ is sufficient to span the entire virtual space of all vectors referenced at lower levels, $\lambda>j$. Therefore, the entire virtual space of a vector referenced at level $\lambda=i$, $i\neq1$ contributes to all higher level localities, $\lambda<i$.

In the case of a column-wise referenced array inside a loop at level $\lambda=i$, one or more columns of an array are spanned during the execution of L_i loop. These columns are usually specified by an outer loop with level $\lambda<i$. The entire virtual space of A' is spanned during one iteration of a loop at level $\lambda=1.2,....i-2$. Thus, the entire virtual space of a column-wise referenced array contributes to localities formed at least two levels higher than the level at which the array is referenced.

Next, the above parameters are used in a more quantitative manner to evaluate the contribution of vectors and arrays to a locality structure. For the convenience of the analysis the cases of vectors and arrays are treated separately.

Vectors

A vector (V) is actually a matrix (M x 1) with M rows and one column. Memory locations in which the elements of a vector are stored constitute the vector's virtual space. The elements of a vector are stored sequentially in a page until the page is filled, and then a second page is used, and so on until all the elements of a vector are completely stored in the virtual storage. A page contains only the elements of one vector, *(homogeneous storage)*. The virtual size of a vector (S_v) is defined as

$$S_v=M/P$$

where P is the page size and M is the number of elements in the vector, assuming that each element is one word long.

Assume that a vector, V_i, is referenced inside a loop at a nest depth λ=i, as shown in Figure 3-5, and V_i's elements are referenced through the indexed variables specified by the current loop, L_i. The locality comprised by loop L_i is the current locality (CL). A vector V_i contributes to the CL as well as to all higher level localities comprised by outer loops, $L_1, L_2, \cdots, L_{i-1}$ with nest depth λ=1,2, $\cdots i-1$, respectively.

Consider the contribution of V_i to the locality at level $i-1$. Assume that L_{i-1} has a range of N_{i-1} iterations. Each iteration of L_{i-1} involves a full execution of L_i. The virtual space of V_i is spanned totally in the time duration of L_i. Therefore, the virtual space of V_i will be spanned N_{i-1} times (the of L_{i-1}). Similarly, at level $\lambda=i-2$ with N_{i-2} iterations, the virtual space of V_i will be spanned $N_{i-2} \times N_{i-1}$ times, thus forming a locality at level λ=i-2. The same analysis applies to all higher level localities. Therefore, a macroscopic view of the virtual space of V_i from any level $\lambda=1,2,\ldots,i-1$ shows that the virtual space of V_i is being referenced repeatedly.

In general, a vector V_i referenced at the current locality level, L_i, contributes to localities at higher levels $L_j, j=1,2,\ldots,i-1$ with its entire virtual size, S_i. For K vectors referenced at CL, each vector contributes to all higher level localities with its virtual size S_i. Let X_i be the size of the current locality and X_j, where j<i, is the size of higher level localities with nest depth $\lambda=j$. Using these notations the contribution of all vectors referenced at CL to all higher level localities is calculated as follows:

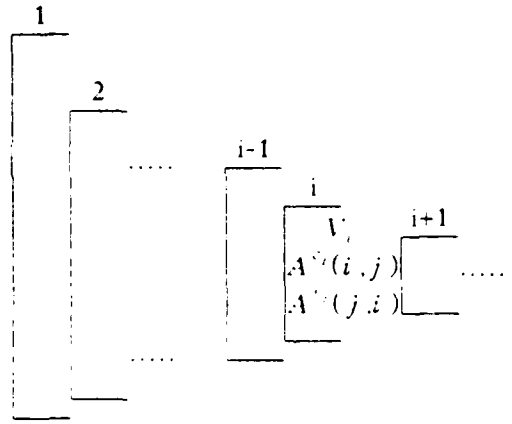


Figure 3-5: Loop structure example

$$X_j = X_j + \sum_{k=1}^{K} S_{v_k} \qquad (3\text{-}2)$$

where j= 1, 2, ..., i-1 and S_{v_k} is the virtual size of the k^{th} vector. K is the number of different vectors referenced inside the CL. For illustration consider Example 3-3.

Example 3-3:

```
Dimension V1(1000), V2(1000)
  DO 10 I=1,1000
    Do 20 J=1,1000
      V1(J)=V1(J)+V2(I)
    20 CONTINUE
  10 CONTINUE
```

Assume that the page size is P=100 words and each vector element is one word. The virtual size of V1 and V2 is 1000/100 = 10 pages each. The code in Example 3-3 adds the sum of the elements of V2 to each element of V1. Each element from the virtual space of V1 (V1(1), V1(2), ... , V1(1000)) is added to one element from V2 (V2(I)) during the execution of Loop 20. All the elements of V1 are referenced during the execution of loop 20, while only one element of V2 is referenced. In other words, the entire virtual space of V1 (10 pages) will be referenced by the time loop 20 completes 1000 iterations. These 10 pages will be referenced again when Loop 10 executes another iteration. By the time Loop 10 iterates 1000 times, the virtual space of V1 will have been spanned 1000 times. Hence, V1 contributes to the locality comprised by loop 10 witn S_{V1}=10 pages. Therefore, if the first level locality comprised by loop 10 is to be considered for allocation, then at least 10 pages must be allocated in order to avoid replacing V1's pages during the execution of Loop 20. The contribution of V1 and V2 to the locality comprised by Loop 20 is discussed next.

The contribution of V_i to the virtual size of CL is determined by the number of distinct vector elements referenced during one iteration of the current loop. The number of distinct elements referenced at level L_i is determined by N, the number of distinct indexed variables used to reference vector elements. The distinct elements of a vector referenced by N indexes can be stored in utmost N pages, depending on the virtual size of a vector and the distribution of N over the vector elements. In Example 3-3, one index, J, is used to reference V1 elements inside loop 20 and I is

used to reference elements in the virtual space of V2. There are only two elements (V1(J) and V2(I)) referenced during each iteration of loop 20. Consider the first iteration of Loop 10, I=1, and the execution of loop 20 (J=1,1000). A reference made to V2(1) is translated to a reference to the virtual address where the first element of V2 is stored. In effect, a reference is made to the first page in the virtual space of V2, $P_1(V2)$. The first page, $P_1(V2)$, which contains the first 100 elements of V2, remains active during the execution of Loop 20 (1000 iterations), since the index I varies only at the level of Loop 10. A reference made to V1(J) is translated to the virtual address of the page containing the element V1(J), depending on the value of J. For example, the first 100 references are made to the first page $P_1(V1)$. The next 100 references ($100<J<200$) are made to $P_2(V1)$, and so on until $P_{10}(V1)$ is referenced. Note that when a new page is referenced, the old one will no longer be referenced until loop 20 is reinitiated by loop 10. Therefore, during the execution of loop 20, V1 needs only one page to be allocated in memory and so does V2. Any extra allocation is redundant.

In general, if the virtual size of a vector is less than the page size, the vector contributes with one page to X_i. However, if the virtual size of a vector is larger than the page size, $S_i > 1$, the number of distinct vector elements referenced at CL comprises an upper bound on the number of distinct pages that could be referenced at CL. The number of distinct vector elements referenced at CL is determined by the number of distinct indexes, N, used to reference a vector.

Figure 3-6 shows a memory representation scheme of a vector. The indexes $I_1, I_2, \ldots, I_N$ are used to reference distinct elements in the form $V(I_1), V(I_2), \ldots, V(I_N)$. The number of distinct

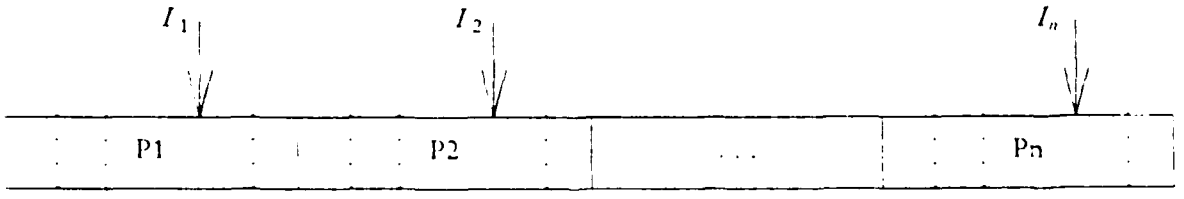


Figure 3-6: A vector's memory representation

indexed variables, N, is used to determine the maximum number of pages that might become active during the execution of CL. Such active pages constitute the body of CL, and N is the virtual size of the locality. Therefore, V_i contributes to the current locality size, with the number of distinct indexes, N pages, or the vector's virtual size, whichever is less. Consequently, the memory request primitive, X_i is given by

$$X_i = X_i + \begin{cases} N+1 & \text{if } N > S_{V_i} \\ S_{V_i} & N \leq S_{V_i} \end{cases} \tag{3-3}$$

A vector is allocated N+1 pages, although the active set of pages contains only N pages. The extra page is used as a buffer to allocate a newly referenced page after N pages have already been referenced. Buffering the new page avoids immediate replacing one of the active N pages. Since the locality of a program contains only N pages, one of the allocated N+1 pages will be idle and, hence, will be a candidate for replacement if a new page is referenced. The underlying assumption, here, is that a least recently used (LRU) or a similar replacement policy is used.

In general, if there are K vectors referenced inside a loop L_i, then the memory requested to allocate K vectors is given by

$$X_i = X_i + \sum_{i=0}^{K} X_{V_i} \tag{3-4}$$

In Example 3-5, each vector has 10 pages in its virtual space. At the first level (loop 10) both vectors need to be allocated entirely since they are referenced at the lower level locality (Loop 20). Hence, the memory allocation primitive at the first level is X=20 pages. The directive inserted at the beginning of Loop 10 would be of the form ALLOCATE (2,20). At the second level, there are

Example 3-5:

```
Dimension V1(1000), V2(1000)
DO 10 J=1,1000
   DO 20 I=1,1000
      V1(I*2); V1(I)
      V2(I); V2(I+1); V2(J);
   20  CONTINUE
10  CONTINUE
```

two indexes, I and I*2, used to reference two elements in the virtual space of V1; hence, N=2 and utmost two memory pages, from the virtual space of V1, are active during the execution of Loop 20. Since N=2 is less than $S_{V\cdot1}$=10, the memory requested to allocate V1 is $X_{V\cdot1}$=2+1=3 pages. Note that if only two pages were allocated to V1, a page will be replaced every 50 iterations and then faulted during the next iteration of the loop. Such extra faults are avoided by using the extra buffering page.

Three indexes, I, $I+1$, J, are used to reference three elements in the virtual space of V2. The three referenced elements could be stored in utmost three pages, N=3. Since $N<S_{V\cdot2}$, the memory required to allocate V2 is $X_{V\cdot2}$=3+1=4 pages. The total memory space required at the second level is N=3+4=7 pages, and the directive at this level has the form ALLOCATE (2,20) else (1,7). Note how ALLOCATE prefers the allocation of 20 (the entire virtual space of V1 and V2) over 7. However, if 20 pages cannot be allocated, 7 pages are enough to avoid thrashing while loop 20 is in control of CPU.

Equations 3-3 and 3-4 are incorporated into a data structure constructed at compile time to estimate the memory requirements of a program. The construction method of such a data structure is discussed later in this section.

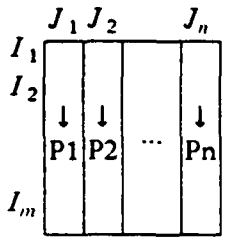
Twodimensional arrays

Depending on their referencing order, arrays can be referenced in a column major order (column wise referenced arrays A^c) or in a row major order (row wise referenced arrays A^r). Both types are discussed in the following subsections.

Column wise referenced arrays

Consider in Figure 3-4, the column wise referenced array $A^c(i,j)$ at level L_i. The column index of A^c, J, remains unchanged during the execution of L_i; J is specified at higher levels $L_1, L_2, \ldots, L_{i-1}$. The value of the row index, I, changes its value at L_i level. Array elements are referenced in the form $A(I,J)$. During the execution of L_i, elements stored in the virtual space of

a: Column Size ⩽ Page Size



b: Column Size > Page Size

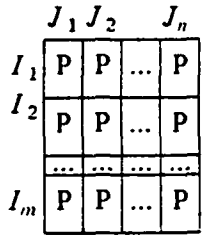


Figure 3-7: Column wise referenced arrays

a column J are addressed using one or more row indexes, $I_1, I_2, \cdots$. Array elements A(I_1,J), A(I_2,J), ... A(I_M,J) could be stored in one page if the column size (S_C) is equal to or less than the page size ($S_C \leqslant$ P) (Figure 3-7a) or in several pages if $S_C >$ P (Figure 3-7b).

In the first case, no matter how many row indexes are used to designate a particular element, only one page could be referenced during the execution of L_i. In the second case, several pages in the virtual space of a column could be referenced during one iteration of L_i. Consequently, the number of row indexes, N_I, used in combination with a particular column index J determines the number of active pages from the virtual space of A^c in the time duration of L_i. Obviously, if the number of pages present in the virtual space of a column is less than the number of row indexes, i.e, $S_C < N_I$, the entire virtual space of a column is active.

Consider Example 3-6, where array A is referenced in a column major order inside the innermost loop of a doubly nested loop. The virtual size of A is $S_A = 1000 \times \frac{100}{100} = 1000$ pages, where the page size P=100 words. The virtual size of each column is $S_C = \frac{1000}{100} = 10$ pages. The sequence of addresses generated during the execution of Loop 20 is shown in Figure 3-8 for J=1. A reference to A(I,1) is translated into a reference to P1 for $1 \leqslant I < 100$, P2 for $100 \leqslant I < 200$, ..., and to P5 for $400 \leqslant I < 500$; i.e., a new page is referenced every 100 iterations of Loop 20. Similarly references to A(I*2,1) generate references to a new page every 50 iterations of Loop 20; i.e., P1 for $1 \leqslant I < 50$, P2

Example 3-6:

```
DIMENSION A(1000,100)
DO 10 J=1,100
   DO 20 I=1,500
      A(I,J); A(I*2,J);
   20  CONTINUE
10  CONTINUE
```

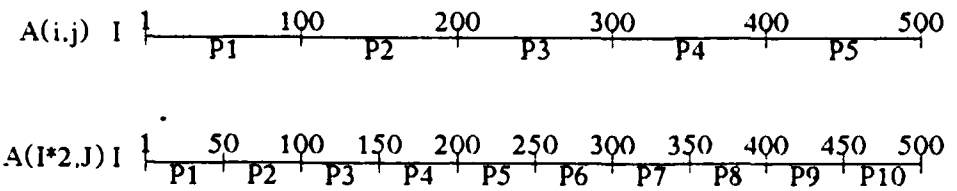


Figure 3-8: Virtual address sequence for A

for $50 \leq I < 100$, P3 for $100 \leq I < 150$, ... and P10 for $450 \leq I < 500$. Figure 3-8 shows that two pages remain active in the time duration of Loop 20, except for the time interval $1 < 50$. These pages are determined by the indexes I and I*2. In principle both elements designated by I and I*2 could be stored in the same page and, therefore, the same page will be referenced twice, or in two different pages and, therefore, two distinct pages will be active. In effect, the number of row indexes used in combination with a particular column, J, gives an upper bound on the maximum number of pages that could be active during one iteration of L_i (Loop 20 in our example). The set of active pages and their time intervals, derived from Figure 3-8, is

$$\{(P1: 1 < 50), (P1,P2: 50 \leq I < 100), \ldots, (P5,P10: 450 \leq I < 500)\}$$

Naturally, more than one column of A^c could be referenced inside L_i. In this case, the number of active pages is defined for each column, J_i, by finding the number of row indexes used in combination with J_i. The maximum number of active pages from the virtual space of an array is found by summing up the numbers found for each column. If the total number of active pages determined in this manner exceeds the number of pages present in the virtual space of an array, the virtual size of the array defines the set of active pages at the current execution level.

The referencing behavior of A^c resembles that of a vector. In fact, an (MxN) array referenced in a column major order can be viewed as a set of N vectors, each vector containing M elements. The memory required to allocate the active pages of a column, X_C is given by

$$X_C = \begin{cases} N_I + 1 & \text{if } N_I < S_C \\ S_C & \text{if } N_I \geq S_C \end{cases} \tag{3-5}$$

The extra page "$N_I + 1$" is used to avoid replacing active pages when a new page is activated as discussed earlier. Memory requirements of a column wise referenced array, X_{A^c}, is defined as the sum of the memory requirements defined for each column, or

$$X_A^c = \sum_{j=0}^{N} X_{C_j} \tag{3-6}$$

where N is the number of columns addressed at level L_i ; $j=0$ means that no array is referenced in a column major order. In general, if there are K arrays referenced in a column major order, the memory required to allocate these arrays at the current level of execution, L_i, is given by

$$X_i = X_i + \sum_{k=0}^{K} X_{A_k^c} \tag{3-7}$$

where $X_{A_k^c}$ is the memory requirement of the k^{th} column wise referenced array.

Next we evaluate the contribution of a column wise referenced array to higher level localities. A column wise referenced array contributes to all higher level localities of levels L_j where j = 1, 2, ..., i-2 with its entire virtual size. The A^c's contribution to the next higher level locality, L_{i-1}, is similar to its contribution to CL, comprised by L_i, because the virtual space of A^c is referenced only once during the execution of L_{i-1}. Whereas, at higher levels, $L_1, L_2, \cdots, L_{i-2}$, the virtual space of A^c is entirely referenced at least once during each iteration of any L_j loop, where $j = 1.2.....i-2$. The memory request primitive at higher levels, defined by column wise referenced arrays is given by

$$X_j = X_j + \sum_{k=1}^{K} S_{A_k^c} \tag{3-8}$$

where j = 1, 2.....i-2 and K is the number of different arrays referenced inside L_j. $S_{A_k^c}$ is the virtual size of the k^{th} array.

Example 3-7 is used to further explain the process of calculating the virtual size of a locality comprised by arrays referenced in a column major order. Two arrays (A1 and A2) are referenced in a column major order inside the innermost loop of a triply nested loop.

Example 3-7:

```
Dimension A1(100,100), A2(400,100)
DO 10 K = 1, 10
   DO 100 J = 1, 100
      DO 1000 I = 1, 400

         A1(I,J) ; A1(I+1,J); A1(I,J+2); A1(I+1,J+2);
         A2(I,J) ; A2(I*2,J); A2(I,J+5); A2(I+2,J+5); A2(M-I,J+5);

      1000 CONTINUE
    100 CONTINUE
 10 CONTINUE
```

A1 and A2 are referenced inside Loop 1000. The contribution of A1 and A2 to the localities defined at level one (Loop 10), level two (Loop 100), and level three (Loop 1000) is evaluated, using Equations (3-5) through (3-8). The virtual sizes of A1 and A2 are given by

$$S_{A1}=\frac{100\times100}{100}=64 \text{ pages and } S_{A2}=\frac{400\times100}{100}=256 \text{ pages}$$

and the virtual size of each column of A1 is $S_{C_{A1}}=100/100=1$ page and each column of array A2 is stored in $S_{C_{A2}}=400/100=4$ pages. The memory requirements of A1 and A2 at loop 1000 level are found as follows. For A1, two columns are referenced inside Loop 1000. Since each column has only one page in its virtual space, there could be only one active page in the virtual space of J and J+2 during the execution of Loop 1000. Therefore, the memory requested to allocate A1 is equal to the number of referenced columns, or X_{A1} = 2 pages.

For A2, each column occupies 4 pages. And there are 2 columns referenced inside Loop 1000 (J and J+5). Two elements n the virtual space of column J are designated by the row indexes I and I*2 (N_j = 2). The memory required to allocate both active pages of J_{A2} is given by Equation (3-5): X_{A1} = N_j +1 = 3 pages. And three elements are referenced from the virtual space of column J+5. These elements are specified by the row indexes I, I+2, and M-I. The maximum number of active pages from the virtual space of J+5 is given by N_j = 3. Hence, the memory space required to allocate these active pages is given by: X_{J+5} = 3+1 =4 pages. The total number of pages required to

allocate A2 at the lowest level (Loop 1000) is X_{A2} = 3+4 =7 pages. Finally, the memory requirement of A1 and A2 at the lowest level (Loop 1000) is X_3 = 7+2 =9 pages, where X_3 is the memory request primitive of ALLOCATE associated with Loop 1000.

When Loop 100 reiterates, a new set of columns from the virtual space of A1 and A2 is specified and the addressed virtual space will change accordingly. However, the amount of memory required at this level does not change. Therefore, the value of X at this level is also 9 pages (X_2 = 9). At the first level (loop 10) the locality size consists of the entire virtual sizes of A1 and A2. During each iteration of the first level locality (loop 10) the virtual spaces of A1 (100 pages) and A2 (400 pages) are totally referenced. At this outer level, all 500 pages will have been referenced 10 times by the time Loop 10 completes execution. Therefore, the memory requirement at level one is given by X_1 = 100+400 = 500 pages.

Finally, ALLOCATE directives with both primitives, priority and memory request, are inserted in the code of Example 3-7:

```
Dimension A1(100,100), A2(400,100)
ALLOCATE (3,500)
DO 10 K = 1, 10
   ALLOCATE (3,500) else (2,9)
   DO 100 J = 1, 100
      ALLOCATE (3,500) else (2,9) else (1,9)
      DO 1000 I = 1, 400

         A1(I,J) ; A1(I+1,J); A1(I,J+2); A1(I+1,J+2);
         A2(I,J) ; A2(I*2,J); A2(I,J+5); A2(I+2,J+5); A2(M-I,J+5);

      1000 CONTINUE
   100  CONTINUE
10  CONTINUE
```

Row wise referenced arrays

A memory representation scheme of a row wise referenced array, A', is shown in Figure 3-9. A row index I_i (i =1,....m) of A' remains unchanged, during the execution of CL in which A' is referenced, whereas the column index J changes its value within the range J_1 and J_N, where N is the number of columns (the second dimension of the array). In Figure 3-9, the arrows point to the direction in which the elements are referenced. With I_i being fixed, the elements

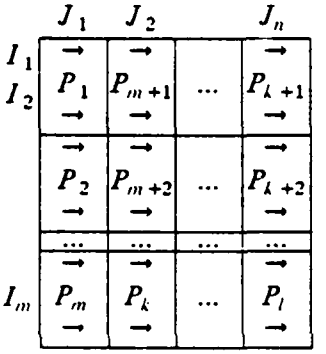


Figure 3-9: Row wise referenced arrays

$A(I_1 J_1), \cdots A(I_1 J_N)$ are stored across pages. If the column size, specified by the first dimension M of the array, is larger than a page size (M>P), then any two elements referenced in a row major order are fetched from two different pages. Two successively referenced elements may be stored in the same virtual page if the virtual size of a column is less than P (M<P).

The contribution of A' to CL, comprised by L_i in Figure 3-5, is determined by the maximum number of pages repeatedly referenced during the execution of L_i. Assume that the elements of the first row, I=1, are referenced during the execution time of L_i. A reference to the element A(1,1) is translated into the address of P_1. The next element A(1,2) will be referenced during the next iteration of L_i, assuming that J is incremented by 1. The page containing A(1,2) is P_{m+1} (Figure 3-9). Every next iteration generates an address to a new page in the virtual space of A'. The referencing pattern at L_i level does not seem to comprise a locality of reference. A referenced page, P_i, may not be referenced more than once in the time duration of L_i, unless the same element is referenced more than once at the same level. The fast changing index, J, spans those elements stored in the virtual spaces of columns J, where J = 1,2.....,N. Therefore, N distinct pages are expected to be referenced in the time duration of L. None of the N pages remains active during the execution of L. Such behavior is unfavored in a virtual memory system, since every iteration of L_i requires a reference to the virtual storage to fetch a new page. A newly fetched page proves to be useful, most of the time, only for that reference. Therefore, if A' is referenced at the outer most level,

then it makes no difference if the entire virtual space of the array is allocated or only one page is allocated in main memory.

However, if A' is referenced at level $\lambda > 1$, then a locality of reference is observed at higher levels. The set of pages referenced during the execution of L_i could be referenced again during the next iteration of L_{i-1} if the row index is varied at this level. In this case the same set of pages remains active until the value of I exceeds the page size limit. At any rate, the number of active pages observed at L_{i-1} level is given by the range of the column index J at the lower level L_i. It has been assumed earlier that the range of J is N. In this case, the number of active pages at L_{i-1} level is N. So far we have considered that only one row index is being used in combination with the column indexes to reference elements in the virtual space of A'. However, several row indexes could be used in any order. In such case, there could be several rows of pages active during the execution of L_{i-1} depending on the relative location of one row index to another. For each row index, I, the memory requirement is given by

$$X_I = \begin{cases} \mathbf{R(J)} & \text{if } \mathbf{R(J)} < S_A \\ S_A & \text{if } \mathbf{R(J)} \geq S_A \end{cases} \tag{3-9}$$

where R(J) is the range of J and S_A is the virtual size of A. In general, if several row indexes are used, the memory required to allocate $X_{A'}$ is given by

$$X_{A'_k} = \sum_{I=1}^{R} X_I \tag{3-10}$$

where A'_k is the k^{th} array referenced in a row major order, and R is the number of row indexes used at level L_i. For K arrays referenced at L_i level, the memory requirement X_i is given by

$$X_i = X_i + \sum_{k=1}^{K} X_{A'_k} \text{ and } X_{i-1} = X_i \,. \tag{3-11}$$

All the elements of A' will get to be referenced in the time duration of L_{i-1} which includes multiple executions of L_i. By this time, all pages in the virtual space of A' will have been referenced. However, only N pages or several sets of N pages, according to Equations (3-9) and (3-10) remain resident in memory during this time, where N is the range of the column index of A'. If L_{i-1} is enclosed by a loop at a higher level L_{i-2}, then all pages referenced at L_{i-1} will be referenced

again during the next iteration of L_{i-2}. Therefore, the entire virtual space of A' contributes to the locality size at L_{i-2} and to all higher levels $L_{i-3}, L_{i-4}, \cdots, L_2, L_1$. The contribution of K arrays referenced in a row major order to X_j at level L_j, where $j = 1, 2, \ldots, i-2$ is given by

$$X_j = X_j + \sum_{k=1}^{K} S_{A_k'} \tag{3-12}$$

where A_k' is the k^{th} row wise referenced array at level L_i. For illustration, consider Example 3-8, where two arrays A_1 and A_2 are referenced in a row major order inside Loop 1000. The nest depth of Loop 1000 is λ=3. The virtual size of A_1 is $S_{A_1} = 1000 \times \frac{10}{100} = 100$ pages. And $S_{A_2} = 200 \times \frac{100}{100} = 200$ pages. The virtual size of each column of A_1 is $S_{C_{A_1}} = \frac{1000}{100} = 10$ pages. And $S_{C_{A_2}} = \frac{200}{100} = 2$ pages. Memory representation schemes of A_1 and A_2 are also shown in Example 3-8. The virtual space of A_1 is organized into 10 rows, each of which contains 10 pages. The virtual space of A_2 has two rows, each of which contains 100 pages. Consider the execution sequence, K=1, I=1, and observe the reference pattern during the execution of Loop 1000, J=1,100. References to A_1 are translated into addresses to the virtual space in which the elements of row I=1 and row I=999 are stored. This virtual space consists of the first and the last rows of pages. During the next iteration of Loop 100, I=2, the same set of pages will be referenced again. References will continue to fall into these pages until I > 100, where the second row and the pre-last row will be referenced. And so at every 100 iterations of Loop 100, a new set of 20 pages is referenced. Therefore, the maximum memory requirement of A_1 at this level is 20 pages. Or, as given by Equations (3-9), and (3-10), the memory requirements of A_1 at the second level is $X_{2_{A_1}} = 10 + 10 = 20$ where the range of the column index is 10, as given in the dimension statement.

For A_2 there is only one row index, I, used at the third level (Loop 1000). Hence, the number of active pages consists of one row (100 pages) from the virtual space of A_2. Each row will remain active for half of the time duration of Loop 100. Therefore, the memory requested to allocate A_2 at level 2 is $X_{2_{A_2}} = 100$ pages. Considering both arrays, $X_2 = 100 + 20 = 120$ pages. The execution

of Loop 100 touches completely the virtual spaces of A_1 and A_2; i.e., all 300 pages are referenced at least once in the time duration of Loop 100.

Now consider the case when Loop 10 continues its execution and K is incremented by 1, K=2. Ignoring the details of reference patterns at Loops 100 and 1000, the virtual spaces of A_1 and A_2 are completely touched once more. This process continues until Loop 10 completes execution. Observing the virtual space of A_1 and A_2 from the first level, the locality of reference seems to cover all 300 pages of A_1 and A_2. The memory requirement at this level is given by $X_1 = 200 + 100 = 300$ pages. ALLOCATE directives are inserted into the code as shown in Example 3-8.

3.1.4.3. Data structure for computing X at compile time

This section presents a method for computing X at compile time. Since program localities exhibit a hierarchical structure, a linked list can be very useful for representing localities at various levels of the hierarchy. When a loop is encountered, a new element is added at the head of the list. All data structures referenced inside a loop are considered as part of the record of a recently created element. When a loop exits, its entry element in the list is deleted and the the contribution of data structures to the locality comprised by the exiting loop is evaluated. Also, the contribution of these data structures to higher level localities, represented by all the remaining elements in the list, is evaluated. The outermost loop is always represented by the element at the tail of the list. When this loop exits, the list becomes empty until another loop construct is encountered. Just prior to a deletion of an element from a list, it should contain the virtual size of the locality comprised by the exiting loop, i.e., the memory request primitive X associated with the current locality.

The use of a linked list data structure (LLDS) facilitates a top down parsing strategy with a back tracking. Back tracking is necessary to compute the contribution of data structures referenced at level L_i to all previously parsed higher level loops $L_1, L_2, \cdots, L_{i-1}$.

Example 3-8:

Dimension $A_1(1000,10)$, $A_2(200,100)$

```
DO 10 K=1,10
  DO 100 I=1,1000
    DO 1000 J=1,100
      If I≤200 then A2(I,J)
      If I≤200 then A1(I,J): A1(1000-I,J)
1000  CONTINUE
100   CONTINUE
10    CONTINUE
```

```
ALLOCATE (3,300)
DO 10 K=1,10
  ALLOCATE (3,300) else (2,120)
  DO 100 I=1,1000
    ALLOCATE (3,300) else (2,120) else (1,120)
    DO 1000 J=1,100
      If I≤200 then A2(I,J)
      If I≤200 then A1(I,J): A1(1000-I,J)
1000  CONTINUE
100   CONTINUE
10    CONTINUE
```

A_1's Virtual Space

	J_1	J_2		J_{10}
I_1	P_1	P_{11}	...	P_{91}
I_2	P_2	P_{12}	...	P_{92}
	...	...	...	...
I_{1000}	P_{10}	P_{20}	...	P_{100}

A_2's Virtual Space

	J_1			J_{100}
I_1	P_1	P_3	...	P_{99}
I_2	P_2	P_4	...	P_{100}

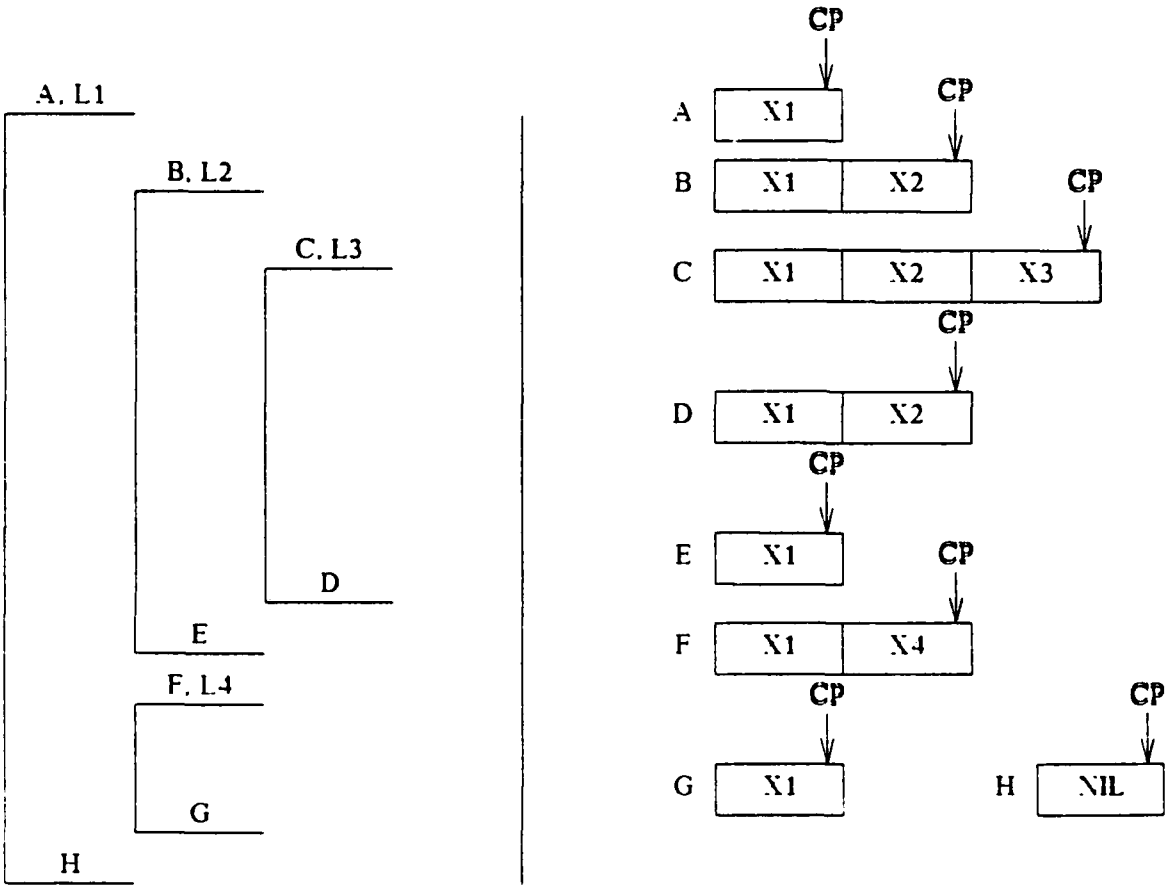


Figure 3-10: Linked list data structure for evaluating primitive X

Figure 3-10 shows the dynamic construction of (LLDS) for evaluating the memory requirements of the loops shown in the figure. A current pointer (CP) always points at the head of the list. Eight parsing stages are shown in the figure. Each stage represents either a beginning or an end of a loop. At stage A, the control statement of the first loop (L1) is encountered. A new element (X1) is created at the head of the list. The current pointer points at X1 which will eventually contain the value of the memory requested by ALLOCATE at level L1, i.e., the virtual size of the locality comprised by L1. The second loop L2 is parsed at stage B and a new entry X2 is added at the head of the list. Now CP points at X2, and will continue to do so until L3 is encountered and X3 is created and added at the head of the list. Loop L3 exits at stage D. At this stage the locality size comprised by L3 is completely computable since all data structures contributing to X3 have been parsed. Also, the contribution of these data structures to X1 and X2 can be evaluated at this point.

The record for X3, which includes the data structures referenced inside L3, is deleted from LLDS. Note that an exiting loop does not enclose any more loops; therefore, its memory requirement is fully determined when it exits. At stage E, X2 is computed. The contribution of data structures referenced inside L3 to X2 has already been determined when L3 exited. Therefore, L2 is treated as if it were an innermost loop, although L2 encloses L3 as indicated by the loop structure. The effect of this technique is similar to unrolling L3 and linearizing the nested structure at L2 level. At stage E, the contribution of data structures referenced inside L2 to X1 is evaluated and X2 is deleted from the list. CP now points at X1. At this stage X1 contains the memory requirements due to L2 and L3. Loop L4 is the only remaining enclosed loop that affects the locality at level L1. Loop L4 is encountered at stage F, where a new element X4 is added at the head of the list. At stage G, X4 is computed and the contribution of L4 to X1 is found. Finally, the memory requirement of the entire loop construct is evaluated at stage H, when L1 exits.

The list data structure described above allows a single top-down parsing scheme. However, a back tracking mechanism is necessary to add the contribution of lower level localities to higher level ones because of the hierarchical nature of localities. Back tracking achieves the same effects of unrolling enclosed loops and linearizing the nested loop structure. Moreover, the LLDS technique transforms the job of back tracking to a simple scan of the list.

Each element of the list is a list structure by itself. A graphic illustration of one element of the list, X_i, is shown in Figure 3-11. A record X has two major fields, one for vectors and one for arrays. The array field has two fields, one for column wise referenced arrays, A^c, and the other is for row wise referenced arrays, A^r. The vector field has several entries, one for each vector referenced at the current level L_i, represented by X_i. Each vector is described by two attributes: the vector variable identifier V_j and its virtual size S_{V_j}. S_{V_j} is used for evaluating the contribution of V_j to higher level localities represented by $X_1, X_2, \ldots, X_{i-1}$. Furthermore, each vector is characterized by a list of distinct indexes used to reference V_j elements at L_i level. The number of entries in the index list determines the maximum number of pages required to allocate V_j at the current level.

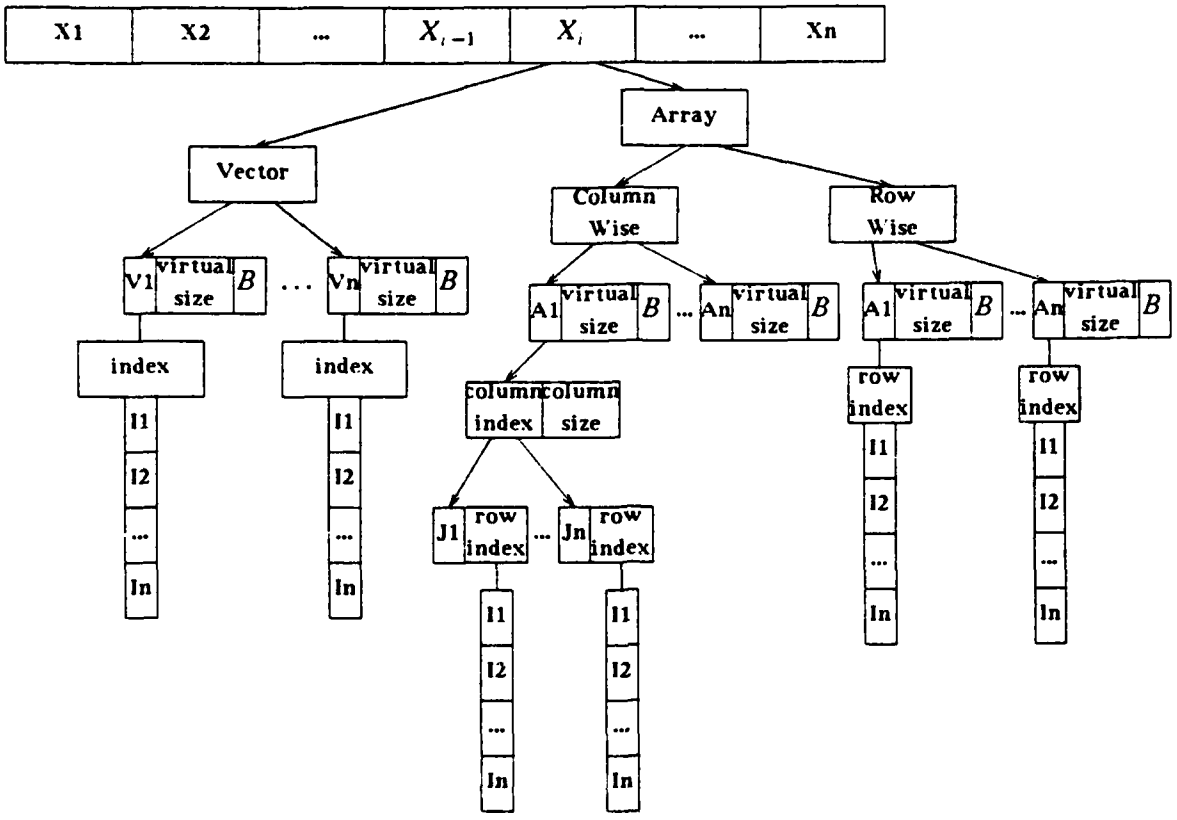


Figure 3-11: Data structure for evaluating X

The B field in the V_i record serves as a boolean variable. The value of B is set to 1 if V_i is not referenced at any lower level, $L_{i+1}, L_{i+2}, \ldots, L_N$. The need for such a boolean variable will shortly be explained.

Each column wise referenced array has several entries, one for each array A_i^c. Each A^c is described by the array identifier A_i, its virtual size S_{A_i}, a boolean variable B similar to the one used for vectors, and a list of the columns referenced at the given level. Each column in the column list is characterized, in its turn, by its virtual size and a list of row indexes used for designating particular array elements. The contribution of any array referenced at L_i to X_i is computed as follows. For each column J_i we find the number of entries N_{J_i} in the list of row indexes which is, then, compared with the value of the column virtual size S_{J_i} stored in the field of the column index record. The least of N_{J_i} and S_{J_i} defines the memory requirement requested to allocate the given

column. The contribution of A_i to X_i is found by summing up the values obtained for each column J_i. The contribution of A_i to X_i is also attributed to X_{i-1}. The array A_i contributes to all higher level localities, represented by $X_1, X_2, \ldots, X_{i-2}$, with the value stored in the virtual size field in A_i record.

A row wise referenced array is described by an identifier A_i, the virtual size of A_i, the boolean variable B and a list of row indexes used at the current L_i level. The value in the virtual size entry is attributed to the memory requirements X1, X2, ..., Xi-2. The number of entires in the row index sublist multiplied by N (the range of the column index or the second dimension of the array) defines the contribution of A_i to X_i, and the next higher level, Xi-1.

At any level of the main LLDS list, there should be only one copy of any array (a vector is a onedimnesional array). This restriction avoids allocating memory to the same array more than once. Assume that an array A is referenced at two levels L_i and L_j where $j < i$; i.e., L_j is higher than L_i. Data structures created for A at X_i level contribute to both X_i and X_j. Data structures constructed at X_j level contribute only to X_j. If data structures for A were kept at both levels, then A would be allocated more memory than it actually requires. Obviously, if the copy associated with X_j is considered and that associated with X_i is ignored, then the memory request X_i, at L_i level, will be underestimated. Hence, data structures created for A at X_i level should be used for computing X_i and X_j. Data structures at X_j are ignored.

The boolean variable B associated with every array referenced at any level is used to enforce the use of *one copy for an array* rule. When a data structure is created for an array A at level X_i, the boolean variable B is set to 1 ($B=1$). The value of B associated with A at all higher levels ($X_1, \ldots, X_{i-1}$) is reset to 0 ($B=0$). The contribution of any array with $B=0$ is ignored, since the contribution of this array has been accounted for at a lower level. Next a procedure is presented for computing X.

```
Procedure (3-1) Compute X;
BEGIN
  Initialize LLDS: LLDS := NIL;
```

```
    Case of encountering a loop L_i DO
    BEGIN
      Create X_i;      {X_i has two fields}
              Vector:  list;
              Array:   (column wise, row wise);
                    Column Wise Arrays: List;
                    Row Wise Arrays: List;
      Initialize the list of vectors (VL): VL:=NIL;
      Initialize the list of column wise referenced arrays: CAL:=NIL;
      Initialize the list of row wise referenced arrays: RAL:=NIL;
      CP := Pointer to X_i;
    End;
    Case of Parsing a vector V_i(I_i) DO
    BEGIN
      IF V_i ∈ X_i
         THEN Updated V_i
         ELSE Create V_i;
    END;
    Case of Parsing an array A_i(I_i, J_j) DO
    BEGIN
      IF A_i is A^c
         THEN IF A_i ∈ X_i
            THEN Update A_i^c
            ELSE Create A_i^c;
         ELSE IF A_i ∈ X_i {A_i is A^r}
            THEN Update A_i^r
            ELSE Create A_i^r;
    END;
    Case of Exiting a loop L_i DO
    BEGIN
      Compute X_i;
      Compute the contribution of data structures at X_i level to X_1, X_2, ..., X_{i-1} levels;
      Reset B =0 for each V_i and A_i encountered at level X_i and any other higher level;
      Delete X_i from LLDS;
    END;
  END. {of Procedure 3-1}
Procedure (3-2) Create V_i;
BEGIN
    Create a new elment (V_i) at the head of the vector list (VL);
    Compute S_{V_i};
    Create index list (IL) for V_i;
    Enter I_i into IL.
END; {of procedure Create V_i}
Procedure (3-3) Update V_i;
BEGIN
    IF I_i is not a member of LI(V_i)
       THEN Add I_i to the index list LI of V_i;
END; {of procedure Update V_i}
Procedure (3-4) Create A_i;
BEGIN
    Create a new element A_i at the head of the list of column wise referenced arrays (CAL);
    Compute the virtual size of A_i, S_{A_i}; {store S_{A_i} in A_i record}
    Create a column index list (CIL);
```

```
   Compute the virtual size of a column S_{C_{A_i}}; {store it in CIL record};
   Enter the column index J_j into CIL;
   Create a row index list (RIL) for J_j;
   Add the row index I_i to RIL.
END; {of procedure Create A sub i sup c}
Procedure (3-5) Update A_i^c;
BEGIN
   IF J_j ∈ CIL
      THEN IF I_i ∈ RIL(J_j)
         THEN Skip
         ELSE Add element I_i to RIL(J_j);
      ELSE
        BEGIN
          Add J_j to the list of column indexes CIL;
          Create a row index list RIL for J_j;
          Add I_i to RIL(J_j);
        END; {of ELSE statement}
END; {of Procedure Update A_i^c}
Procedure (3-6) Create A_i^r;
BEGIN
   Create A_i element at the head of the list of row wise referenced arrays RIL;
   Compute the virtual size of A_i; {store S_{A_i} in A_i record}
   Create a row index list RIL for A_i; {RIL:=NIL}
   Add the row index I_i at the head of RIL;
END; {of Procedure Create A_i^r}
Procedure (3-7) Update A_i^r;
BEGIN
   IF I_i ∈ RIL(A_i)
      THEN Skip
      ELSE Add I_i at the head of RIL(A_i);
END; {of Procedure Update A_i^r}
```

Consider the following notations and definitions which are necessary to define a procedure for evaluating X_i when the corresponding loop L_i exits. The length of a list L is the number of elements in the list. Each vector is associated with a list of row indexes: the length of this list is denoted by $L(V_i), i = 1.....N$, where N is the number of vectors or $N = L(VL)$. Each column index of A^c has a list of row indexes: the length of this list is denoted by $L(J_i), i = 1.....K$ where K is the length of the list of column indexes, $K = L(CIL)$. The number of A^c is given by M the length of A^c list, $M = L(CAL)$. Each row wise referenced array A_i^r has a list of row indexes: the length of this list is denoted by $L(A_i), i = 1.....S$, where S is the length of the list of A^r, $S = L(RAL)$. The range of the column index of a row wise referenced array is denoted by $R(A_i^r)$. Using these notations, the following function can be used to compute X_i:

$$X_i = X_i + \sum_{i=1}^{N} \min(L(V_i), S_{V_i}) +$$

$$+ \sum_{m=1}^{M} \{\min(\sum_{k=1}^{K} \min(L(J_k), S_{C_m})) \cdot S_{A_m^c}\}$$

$$+ \sum_{s=1}^{S} \min(L(A_s) \times R^c(A_s) \cdot S_{A_s}) \quad . \tag{3-13}$$

The terms in Equation (3-13) represent the contribution of vectors, column wise referenced arrays and row wise referenced arrays, respectively. This contribution is added to what has been already stored in X_i field, due to contributions from lower levels.

The contribution of vectors and arrays to higher levels is given by the following two formulas:

$$X_j = X_j + \sum_{i=1}^{N} S_{V_i} + \sum_{m=1}^{M} S_{A_m^c} + \sum_{s=1}^{S} S_{A_s^r} \tag{3-14}$$

and

$$X_{i-1} = X_{i-1} + \sum_{i=1}^{N} S_{V_i} + Q \tag{3-15}$$

where Q is the last two terms in Equation (3-13).

3.1.5. Automatic insertion of ALLOCATE at compile time

ALLOCATE is inserted just before the beginning of each loop comprising a locality. The two primitives of ALLOCATE, P and X, are computed and assigned to each loop according to Algorithm 3-1 for P and Procedure 3-1 for computing X. It would have been very simple to insert ALLOCATE at the beginning of each loop, once P and X are evaluated, if ALLOCATE exhibited a linear structure. Because of the hierarchical structure of ALLOCATE, the primitives of higher level localities are carried into all subsequent lower level localities. Therefore, the mechanism to be used for inserting a directive at a particular level should be able to memorize the primitives associated with all levels enclosing the current level. The memory capacity should be at least equal to the nest depth of the currently parsed loop. A suitable data structure for implementing such a mechanism is a stack or a linked list.

ALLOCATE directives are inserted using a stack data structure as follows. When a loop L_i is encountered, its primitives (P_i, X_i) are pushed to the top of the stack. When a loop L_i exits, the (P_i, X_i) pair at the top of the stack is deleted. At any parsing level, a directive's parameters consist of all the elements in the stack ordered from bottom to top and separated by the word "else." The directive inserted at the beginning of L_i has the form

$$ALLOCATE\ (P_1, X_1)\ else\ (P_2, X_2)\ else\ \cdots\ else\ (P_i, X_i)$$

Linked list implementation is similar to stack's, since a linked list is a form of stack. Besides simulating the hierarchical nature of ALLOCATE, stack implementation facilitates a single top down parsing scheme without backtracking. Algorithm 3-2 automatically inserts ALLOCATE at compile time into a program's compiled code.

Algorithm 3-2: Insert ALLOCATE at Compile Time;
Initialize the directive's stack DS;
Parse {*until the end of the program*}
Case of encountering a loop L_i **with primitives** (P_i, X_i) **DO**
BEGIN
PUSH (P_i, X_i) **at the top of DS;**
FORM the directive
$ALLOCATE\ (P_1, X_1)\ else\ \ldots\ else\ (P_i, X_i)$
{starting from the bottom of DS until the top of DS}
INSERT the directive right before the beginning of L_i**;**
END; {*of case statement*}
Case of exiting a loop L_i **DO**
DELETE the pair (P_i, X_i) **from the top of the stack;**
END. {*of Algorithm 3-2*}

An example using Algorithm 3-2 is shown in Figure 3-12. A loop construct with a maximum nest depth λ=3 is used in Figure 3-12 to illustrate the operation of Algorithm 3-2. The primitives P_i, X_i are assumed to be known for each of the four loops. The directives are inserted as shown at the beginning of each loop. The stack is updated upon encountering of a loop begin control or end control statements. When Loop1 is encountered, the (P_1, X_1) pair is pushed at the top of the stack. The directive at the beginning of Loop1 has the form ALLOCATE (P_1, X_1). Next, Loop2 is encountered and (P_2, X_2) pair is pushed at the top of the stack. The directive at this point has the form

$$\text{ALLOCATE } (P_1, X_1) \text{ else } (P_2, X_2).$$

At stage 3, Loop3 is encountered and the pair (P_3, X_3) is pushed to the top of the stack. The

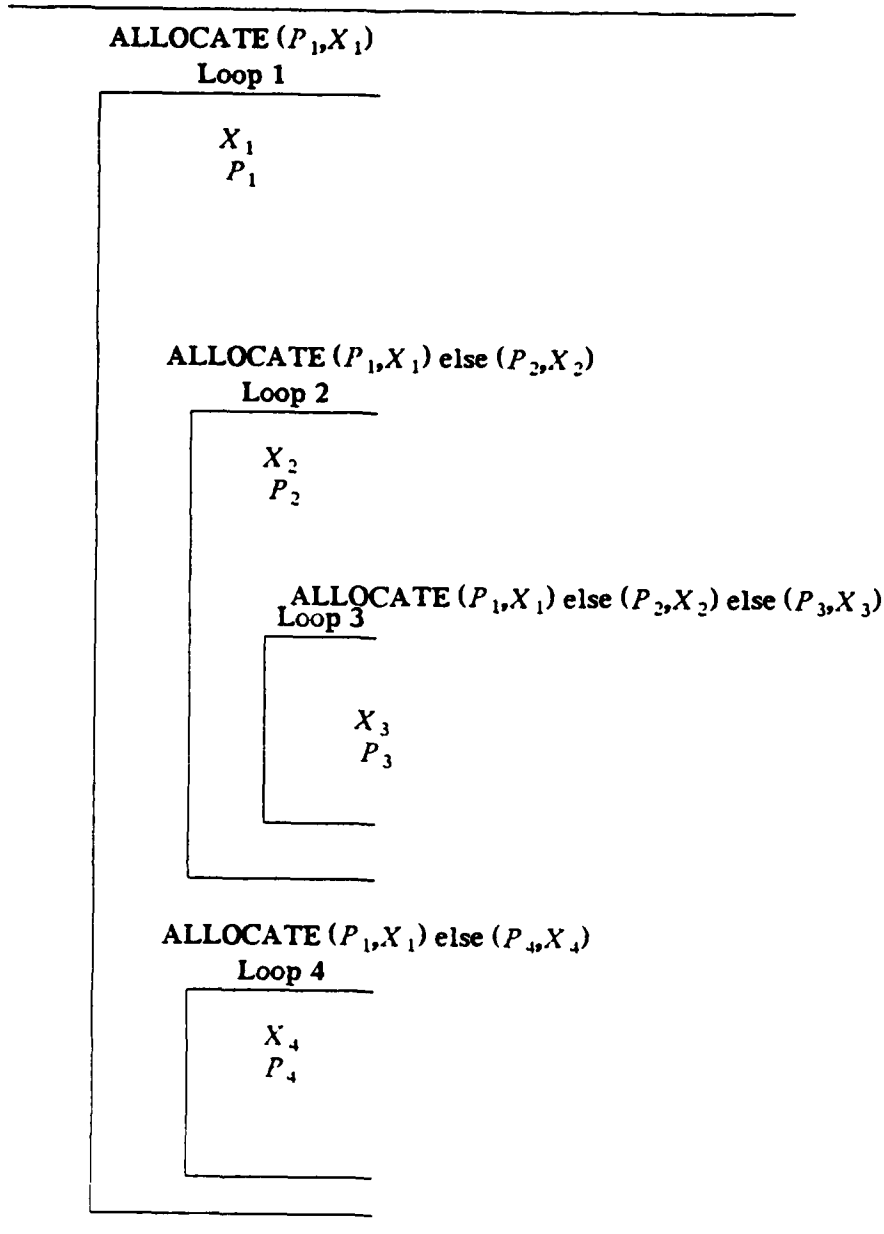


Figure 3-12: Example of algorithm 3-2

directive inserted at the beginning of Loop3 has the form

$$\text{ALLOCATE } (P_1,X_1) \text{ else } (P_2,X_2) \text{ else } (P_3,X_3).$$

When Loop3 exits, the pair (P_3,X_3) is removed from the top of the stack. Similarly, upon exiting

Loop2, the pair (P_2,X_2) is removed. Loop4 is, then, encountered and the pair (P_4,X_4) is pushed at the top of the stack. The directive inserted at the beginning of Loop4 has the form

$$\text{ALLOCATE } (P_1,X_1) \text{ else } (P_4,X_4).$$

Note that Loop 4 is enclosed by Loop 1. The pairs (P_4,X_4), (P_1,X_1) are deleted upon exiting Loop 4 and Loop 1, respectively; the stack remains empty until another loop structure is encountered.

3.2. LOCK and UNLOCK Directives

LOCK is used to prevent particular pages from being paged out of memory by the replacement policy. UNLOCK is used to relase these pages. LOCK and UNLOCK have been used as system facilities by VAX/VMS and UNIX operating systems. Abaza [1] measured the effectiveness of using LOCK and UNLOCK under VMS. His results show that the behavior of some numerical algorithms can be drastically improved, if LOCK and UNLOCK under VMS are properly used. However, in these systems the problem of locking and unlocking particular pages is still a user rather than a system problem. A user is supposed to have adequate knowledge of the behavior of his program. In particular, he should be able to identify those pages which are needed mostly in memory so he can order them locked in memory.

In this study, pages to be locked in memory are identified automatically at compile time. As in the case of an ALLOCATE directive, the cases of vectors and arrays are considered separately. In general, a page may be a candidate for locking if it is located in an *intra-locality* transition period. Intra-locality transition periods occur within a hierarchical locality structure, whereas inter-locality transitions occur between two successive hierarchical locality structures. Using source level code notations, intra-locality transitions are caused by references to array data structures in between two successive loop start control statments. Let L_i refer to the beginning of a loop in a multi-nested loop structure and L_{i+1} refer to the beginning of the next loop. Intra-locality transition pages are those pages referenced in between L_i and L_{i+1}.

A page referenced in an intra-locality transition period does not contribute to localities formed at the next lower levels $L_{i+1}, L_{i+2}, \cdots$. Intra-locality transition pages, on the other hand,

are included in all higher level localities, $L_1, \cdots, L_i$. Further illustration is presented in Example 3-9.

Example 3-9

```
DO 10 i=1,N
   V1(i)
   DO 100 j=1,M
      V1(j)
   100  CONTINUE
10   CONTINUE
```

In Example 3-9, a page of vector V1 designated by the virtual address of V1(i), $P_{V1(i)}$, is referenced in the transition period between Loop 10 and Loop 100. This page remains idle as long as Loop 100 is in execution. However, it is reactivated when Loop 100, after M iterations, returns control to Loop 10. Therefore, locking $P_{V1(i)}$ in memory avoids the need to page it into main memory every time loop 10 executes. Note that if the request generated by an ALLOCATE directive associated with loop 10 (ALLOCATE $(2,S_{V1})$) is granted, locking a page from the virtual space of V1 has no significance.

Thus far, a LOCK directive may have the following form with one primitive:

$$LOCK\ (Y_1, Y_2, \cdots, Y_n)$$

where Y_i is a particular virtual page. Once LOCK is executed by the CPU, a request is made to the operating system to lock into memory those pages identified by the virtual addresses $Y_1, Y_2, \cdots, Y_n$. Pages are unlocked, or released, by an UNLOCK directive which has the following form:

$$\text{UNLOCK}\ (Y_1, Y_2, \cdots, Y_n).$$

LOCK is inserted inside th loop and UNLOCK is inserted at the end of L_i. See Example 3-10.

Example 3-10:

```
DO 10 i=1,N
   V(i):
     LOCK P_V(i)
      DO 100 j=1,M
         V(j)
      100  CONTINUE
10   CONTINUE
UNLOCK P_V(i)
```

In a multi-nested loop structure, pages could be locked at various levels of a locality structure. Therefore, it is possible that a program would be running with its lowest level locality (P=1) while some pages belonging to higher level localities are being locked in main memory as a result of executing a LOCK directive. In case of high memory contention, a program should be allowed to run only with its lowest level locality. Partial swapping, introduced previously for ALLOCATE, guarantees that higher level localities are not allocated when a program's request with $P=1$ cannot be granted. In a similar fashion, the operating system should be allowed to unlock a previously locked page, even before it is released by an UNLOCK directive. Since pages can be locked at various levels of a locality hierarchical structure, a priority index P can be used to define the priority of releasing a page by the operating system before it is released by UNLOCK. For this purpose a *priority index primitive* is introduced into the LOCK directive:

$$LOCK\ (P, Y_1, Y_2, \cdots, Y_n).$$

Pages locked at the lowest level of a locality structure should be released last, since they are invoked more frequently than those referenced and locked at higher levels. Therefore, pages locked at lower levels of the locality hierarchical structure should have a higher priority than those at higher levels. To be consistent with the priority index used for ALLOCATE, smaller P values are used to denote a higher priority. In other words, pages locked with larger P values are released before pages locked with smaller P values. Priorities are assigned to loops in the same way as for an ALLOCATE directive; see Algorithm 3-1. A directive may have in principle a priority $P=1$, associated with the innermost loop. However, in practice such a directive is never used because the memory requirement of the innermost loop, defined by the ALLOCATE directive, is always granted. For further illustration, consider the example in Figure 3-13.

The maximum nest depth of the loop construct in Figure 3-13 is 3. The priorities assigned to L1 and L2 are P=3 and P=2, respectively. The value P=1 is the priority of L3. Inside L3 no pages should be locked, since the locality comprised by L3 is allocated by the ALLOCATE directive with P=1. Assume that the ranges of loops L1, L2, and L3 are K, N, and M, respectively. Each Y_i page is referenced, at level L2, at least N times more than any X_i page, referenced at level L1.

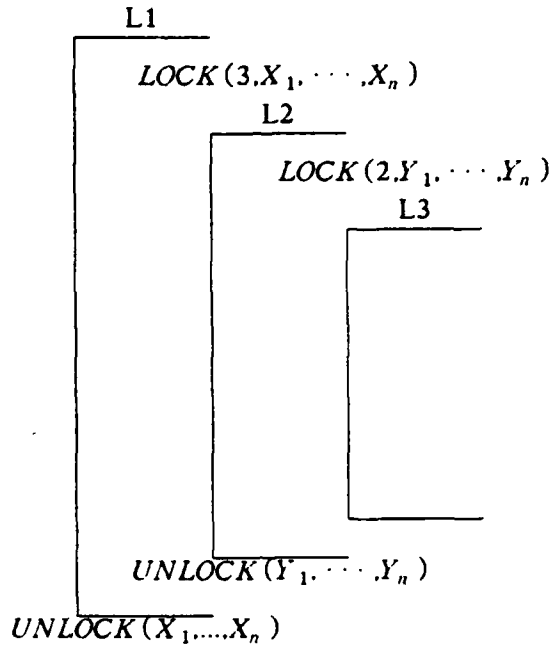


Figure 3-13: Example of LOCK and UNLOCK directives

For this reason, a page locked at a higher level, L1 in this case, should be released before a page locked at a lower level, L2 in our example.

Inserting LOCK at compile time is very simple since LOCK does not exhibit a hierarchical structure as ALLOCATE does. Algorithm 3-3 is used for automatic insertion of LOCK and UNLOCK directives.

Algorithm 3-3: Insert LOCK and UNLOCK directives;
CASE *of encountering a loop* **DO**
P = P assigned to current loop;
Y_i = Page to be locked (i= 1,2, ... , n);
IF P ≠ 1 THEN INSERT
1- LOCK *after the loop BEGIN control statement;*
2- UNLOCK *after the loop END control statement;*

Pages to be locked by LOCK are either vector or array pages as discussed earlier for ALLOCATE. For the case of a vector, V, any page referenced at some level L_i is likely to be rereferenced after the execution of the L_{i+1} loop. A reference to a vector element V(j) is translated to a refer-

ence to a virtual page $P_{V(j)}$. If more than one vector element is referenced at a particular level, using more than one index, then it is possible that more than a page needs to be locked, depending on the value of the indexed variable. Therefore, a page to be locked is identified by the referencing index, j. For example, if a vector is referenced as V(j), V(j1), V(j2), then the page(s) containing these three elements is locked. At compile time, a candidate page for the LOCK directive is identified by the vector name identifier and the vector's indexed variable; no address translation is assumed at compile time. At run time, a reference to a vector element V(j) is translated into the virtual address of the page $P_{V(i)}$ storing the element V(i).

The fact that OS can release a locked page before UNLOCK does so, gives LOCK a *soft* property. LOCK's soft property can be incorporated into the partial swapping mechanism. This feature of the swapping mechanism further supports the property of redistributing memory space among processes in cases of high memory contention.

For arrays referenced in a row major order, a referenced page at L_i level is unlikely to be rereferenced after the execution of L_{i+1} unless the page size is larger than the column virtual size of the array, where two successive row elements may be stored in the same page. Therefore, a page of A^r may be locked only if the column virtual size is less than the page size, where a page to be locked is of the form $P_{A(i,j)}$ where i is the row index and j is the column index.

Arrays referenced in a column major order, A^c, are similar to vectors. Each column, in fact, resembles a vector. Therefore, for each column the distinct row indexes determine the virtual pages that may be referenced at a given level. A page to be locked is identified at compile time as $P_{A(i,j)}$ where i is the row index and j is the column index.

The implementation of LOCK is fairly simple. A lock bit (LB) is associated with each page. When a request is generated to lock a page Y_i into memory, the lock bit of Y_i is set to one, $LB(Y_i)=1$. The replacement policy avoids replacing any page with LB=1. The partial swapping mechanism searches for pages with LB=1 for unlocking them when initiated by a running process.

A list data structure (LLSD) similar to the one designed for ALLOCATE can be used to identify those pages which should be locked at each level. Once a new loop L_i is parsed, a new entry X_i is created and appended at the head of the list; LLSD is initially empty. Upon exiting a loop L_i, the list of virtual pages found at this level is assigned to LOCK directive and the element X_i is deleted. If the exiting loop has P=1, no LOCK directive is inserted. The data structures created for each element are similar to those described for ALLOCATE. The main difference is that for ALLOCATE the number of distinct pages that could be referenced at a particular level is of primary concern, whereas for LOCK the particular pages referenced at a given level are of primary concern. Moreover, the data structures associated with an element do not contribute to other elements in the list; therefore, no back tracking is necessary. An example is given in Figure 3-14 where the primitives of LOCK are further explained.

```
DO 10 i =1,N
    V(i); A(1,i)
        DO 20 j =1,M
            V(j)
                DO 30 k =1,R
                    A(k,j)
                    V(k)
                30 CONTINUE
        20 CONTINUE
        A(i,1)
        DO 40 j =1,M
            A(j,i)
        40 CONTINUE
10 CONTINUE
```

```
DO 10 i =1,N
    V(i); A(1,i)
    LOCK(3,V(i),A(i,1))
        DO 20 j =1,M
            V(j)
            LOCK(2,V(j))
                DO 30 k =1,R
                    A(k,j)
                    V(k)
                30 CONTINUE
        20 CONTINUE
        UNLOCKV(j)
        A(i,1)
        DO 40 j =1,M
            A(j,i)
        40 CONTINUE
10 CONTINUE
UNLOCK(V(i),A(i,1))
```

Figure 3-14: Example using LOCK and UNLOCK directives

3.3. Subprogram Sequence Control Under CD

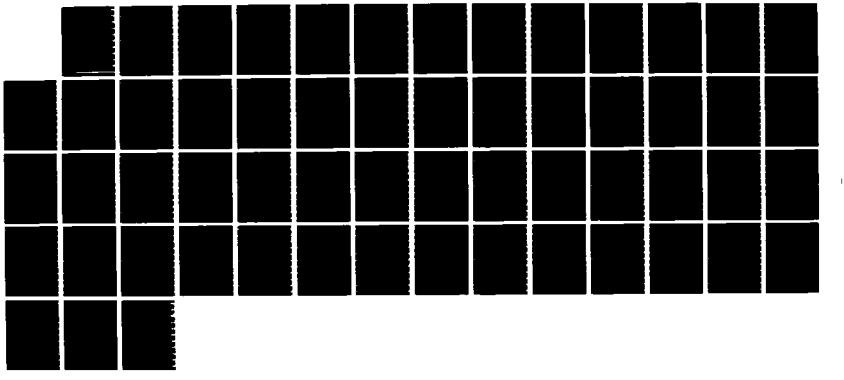
In this section our concern is with mechanisms for controlling memory allocation when a subprogram is called. Programs are usually hierarchically structured into a main program and subprograms. Each subprogram may call another subprogram and so forth. The simplest control structure of subprograms can be explained by the *copy rule* (CR). The effect of a subprogram CALL statement is the same as would be obtained if the CALL statement were replaced by a copy of the body of the subprogram before execution. Viewed in this way, a locality may be comprised partially by the calling program and partially by the called subprogram. Memory directives are inserted into the program code after subprogram CALL statements have been substituted by the subprogram body. During execution, a call to a subprogram will have no effect on the current memory allocation unless the called subprogram generates a new memory directive with a new memory allocation request.

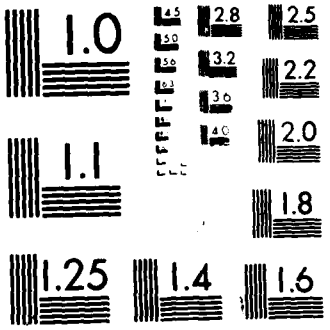
The copy rule could, explicitly, be applied and the body of a called subprogram be copied *in-line* only if the subprogram is very short. Otherwise, a subprogram call is eliminated in principle, not in practice. Identifying program localities under the copy rule is a complex problem, since subprograms can no longer be considered separate entities which comprise separate locality structures. Moreover, the depth of a locality hierarchy is increased by as much as the depth of subprogram hierarchical structure. Another major drawback of CR technique is that subprograms can not be recursive. However, recursion is a common characteristic of many algorithms which naturally leads to recursive subprogram structures. Although our program model in this thesis is FORTRAN programs which do not support recursion, it is desired to extend the application of CD to other languages supporting direct or indirect recursion.

In order to simplify the process of directive insertion, a subprogram CALL statement should be treated as a regular statement without affecting the current locality structure. Moreover, a subprogram, when compiled, should be considered as a separate entity consisting of its own locality structures. Finally, it is desired to allow recursive subprogram calls rather than just simple

AD-A171 803 COMPILER DIRECTED MEMORY MANAGEMENT FOR NUMERICAL PROGRAMS(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB M I MALKAWI AUG 86 UILU-ENG-86-2229 2/2

UNCLASSIFIED N00014-84-C-0149 F/G 9/2 NL





CALL-RETURN subprogram statements. These goals can be achieved using the *activation record* technique.

Activation record technique

Under the activation record technique, memory directives are inserted in the usual manner at compile time. Subprogram CALL statements are treated as regular code statements having no effect on the current locality structure. Locality structures comprised by the called subprogram code do not contribute to the locality structures of the calling program. In effect, each subprogram is considered a separate locality entity.

At execution time, when a subprogram is activated due to a CALL statement the calling subprogram or main program is temporarily halted. The memory allocation previously set by directives generated at the calling program level may be altered by a directive generated during the execution of the callee active subprogram. When the execution of a subprogram is completed, execution of the calling program resumes at the point immediately following the call of the subprogram. The memory allocation at this point should be similar to the memory allocation at the point of executing the CALL statement. The activation record technique is used to keep records for memory directives' primitives for each subprogram as long as it remains in execution. A subprogram remains in execution until it returns control to the calling subprogram. The CPU is always controlled by an active subprogram.

At the time of a subprogram call, a new activation record is created for the newly activated subprogram, which is subsequently destroyed upon its return. A simple central stack may be used to store the activation records of all subprograms in execution which have not returned yet. The last item created on the stack must be the first item to be deleted. Similarly, the first item created on the stack is the last one to be deleted. The implementation of the subprogram call and return proceeds as follows. At the start of program execution, a large storage is reserved for the central stack. The activation record for the main program is allocated at one end of the block. This becomes the bottom of the stack.

When a subprogram A is called, a storage for its activation record is allocated adjacent to that of the main program's activation record. If A calls B, B's activation record is allocated adjacent to A's. If B calls C, C's activation record is allocated adjacent to C's, and so on. When C terminates and returns control to B, C's storage is deleted, and then B's when B returns, and so on. The central stack implementation for a series of subprogram calls and returns is shown in Figure 3-15.

Each activation record contains several data objects. One data object is used to store the *return address* of a subprogram which can be thought of as a pointer pointing at the previous activation record in the central stack. Return address values form a linked list that links together the activation records on the central stack in the order of their creation. The current environment pointer

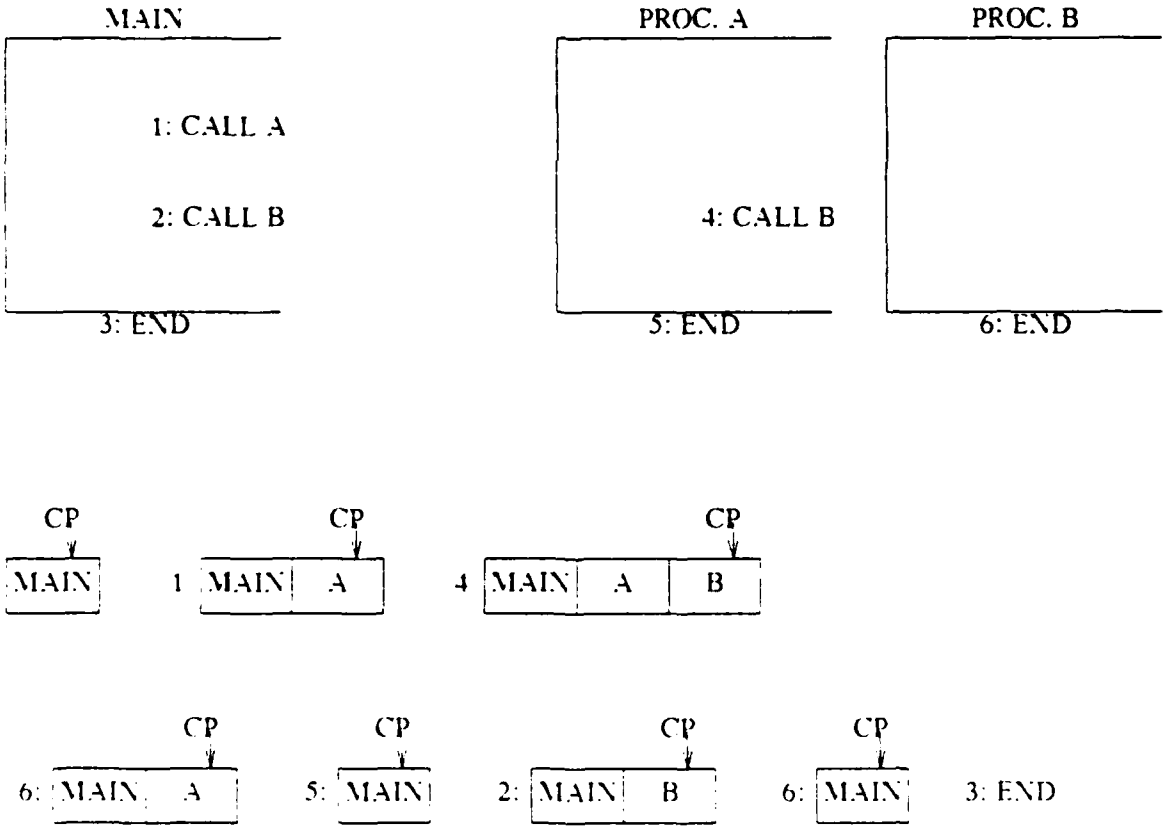


Figure 3-15: Use of a central stack of activation records

(CP) is constantly updated to point at the "top" activation record in the stack. From the return address value in the return point, the second activation record in the stack may be reached. From the return address value of this record, the third activation record can be reached, and so on. At the end of this chain, the last link leads to the activation record for the main program. This chain is called a *dynamic chain* because it chains together subprogram activations in the order of their dynamic creation.

Our main concern in this study is with the *directive primitives (DP)* data object. The directive primitives data object is used to store the values of the (P,X) pair used by the ALLOCATE directive. The entries of an activation record are shown in Figure 3-16. The current memory allocation of a program is determined by the values of the (P,X) pair of the "top" activation record specified by CP. While the new subprogram is executing, the contents of P and X are constantly changing as new directives are executed. When a subprogram terminates, its activation record is deleted together with its data objects. Now CP points at the second activation record in the stack. A previously terminated subprogram resumes execution by restoring the values of (P,X) pair, among other data objects recorded at the time of executing a CALL statement. The memory allocation of a program is determined by the values of the (P,X) pair obtained from the activation record at the top of the stack, pointed at by CP.

When a subprogram A calls subprogram B (executes CALL B statement) the directive primitives entry of A's activation record contains the values P_A and X_A. B executes for a while and then

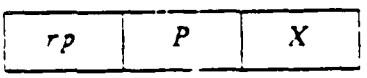


Figure 3-16: Activation record entries

terminates. A then resumes its execution, requesting the allocation of X_A pages with a priority P_A. The request is satisfied if X_A pages can be allocated from the free page pool in main memory. It is possible that X_A pages can not be allocated, although subprogram A was running with X_A pages at the time of its interruption. In such a case, OS invokes the swapper if $P_A = 1$. Otherwise, the execution continues with the current allocation until the next directive is received with a new pair (P, X).

Rather than storing only one (P, X) pair in the activation record, it is more effective to store the set of pairs $(P_1, X_1), (P_2, X_2), \cdots, (P_n, X_n)$ specified by the argument list of ALLOCATE associated with the lowest level locality. When subprogram A resumes its execution after the termination of subprogram B, the activation record of A is searched for a pair P_i, X_i that can be allocated. The first pair to be tried for allocation is P_1, X_1 and then P_2, X_2, and so on until $P = 1, X$ is reached. Note that $P_1 > P_2 > \cdots > P_n$ and $X_1 > X_2 > \cdots > X_n$. This scheme avoids the need to wait for the arrival of a new directive when the current directive entry pair (P, X) can not be allocated. Moreover, it reduces the cost of processing a directive as will be discussed in the next section. An example is shown in Figure 3-17. A multi-nested loop structure with a maximum nest depth of 3 is shown in the figure with ALLOCATE directives inserted at the appropriate levels (ALL stands for ALLOCATE). Memory request primitives, X, are arbitrarily assigned. Note that the number of (X, P) pairs in the directive entry is limited by the maximum depth of the loop structure comprising the current locality. The activation record of A is dynamically updated. At stage 1, the pair $P = 3, X = 100$ is stored in the record. At stage two, the second parameter of the directive is entered into the activation record. At stage 3, the activation DP entry is filled. At any time during the execution of A, the memory space allocated to A is given by one of the (P, X) pairs in the activation record.

Recursive calls to a subprogram is simply implemented by creating a new activation record for a subprogram every time it is called. The size of the central stack may become too large due to an increased number of activation records created for a recursively called subprogram.

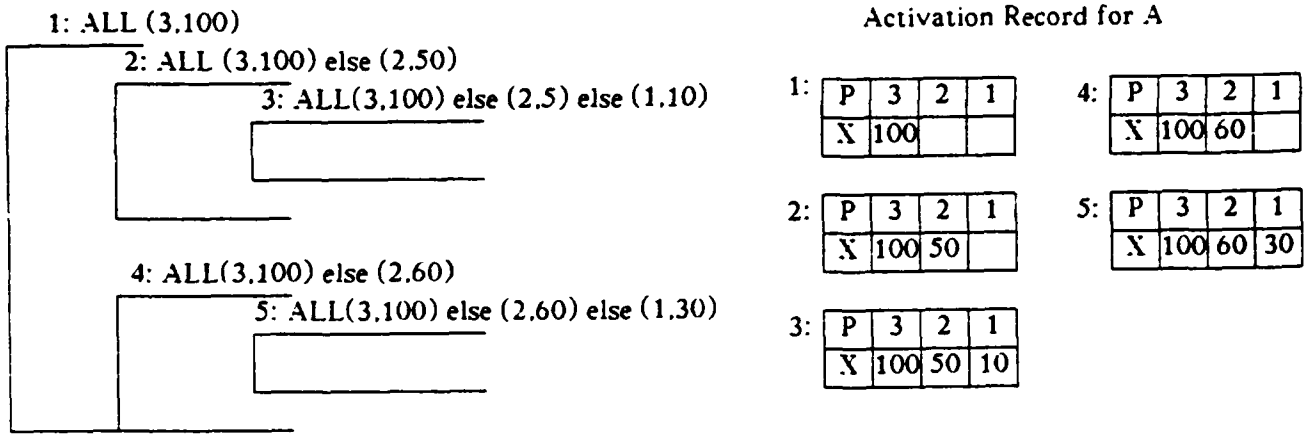


Figure 3-17: Example of subprogram sequence control

3.4. Cost of CD

There are two types of cost associated with CD. The first one is the cost of inserting directives at compile time. The second one is the cost of executing a directive.

Compile time cost is less severe because directives are inserted only once. This cost can be limited by having the directives inserted only into syntactically error free programs. This restriction is aimed at reducing the number of times a compiler has to insert the directives into the program code. The actual cost of inserting memory directives at compile time is not measured in this study. This problem is left for further research and studies.

This section elaborates on the cost of CD at execution time. The cost of CD at execution time is the cost of executing the directives ALLOCATE, LOCK and UNLOCK. Our concern here is with the overhead due to multiple executions of a directive located inside a loop. We will also discuss the overhead due to the execution of "else" conditional statements incorporated by ALLOCATE, since a conditional statement is a time-consuming operation compared to regular evaluating expressions. These two factors contributing to the cost of CD at execution time are discussed next.

The structure of ALLOCATE can be relaxed to exclude the conditional statement "else," thus giving ALLOCATE the simple form ALLOCATE (P,X) where P and X are the primitives associated with a loop. However, it is necessary that ALLOCATE reflects the hierarchical nature of a

locality structure and to respond to the constantly changing memory status of the system due to multiprogramming interaction. One way of preserving the hierarchical structure of ALLOCATE is to use a multiple DP entry for the activation record discussed in the previous section. When a directive is executed, the values of the (P,X) pair are stored in the activation record in a descending order. When a second directive is executed the values of its (P,X) pair are stored in the activation record, and so on until the directive at the lowest locality level with P=1 is executed; at this point the DP entries are filled.

When a program is allocated a time slice, OS examines the activation record at the top of the central stack. The pairs (P,X) are tried for allocation in a descending order. If OS fails to allocate the first pair, it tries the second one, and so on, until the last one is reached. The swapper is invoked upon failing to allocate the pair $(P=1,X)$ as explained in Section 3-1-3. Note that the conditional statement is now transferred to the OS level of execution, where OS checks the values of the activation record and compares them with the available free memory space.

Multiple execution of a directive is caused by executing a directive located inside a loop. Obviously, the directive is treated as a regular instruction which has to be executed, unless otherwise stated. Such multiple execution of a directive adds to the cost of CD, especially when the memory status has not changed since the last time the directive was executed, in which case the execution of a directive is a mere overhead. Using a multiple directive point entry in the activation record and the relaxed form of ALLOCATE proves to be useful in reducing the number of times a directive is executed. A directive inserted at a higher level needs not be executed at lower levels because its primitives have already been stored in the activation record. However, a lower level directive, although relaxed, still has to be executed every time the loop containing the directive iterates.

The optimal solution to this problem is to move all the directives outside the loop structure. This can be done either at compile time when the directives are inserted, or at run time when the directives are first executed. Eventually, all the directives of a loop structure will be stored in the activation record. Therefore, if the removal of a directive is to take place at run time, then once a

directive is processed, its primitives are stored in the DP entry of the activation record and the directive is removed from the program code.

The cost of executing the directives in their original form, without relaxation and without using activation records with multiple data entry, is measured in this study. The results are reported in the next chapter.

3.5. Summary and Conclusions

We have presented in this chapter a compiler directed memory mangement policy (CD). Three memory directives, ALLOCATE and LOCK and UNLOCK are inserted at compile time into the program's source code. When a directive is executed by the CPU during execution time, it generates a request to OS to allocate X number of pages or to lock into memory a particular page. We have developed algorithms for inserting directives, automatically, at compile time. These algorithms utilize source level code information to identify program localities and to evaluate the size of these localities.

We have also treated the problem of subprogram control sequence using the activation record technique. A subprogram may be defined as a subroutine, a function or a procedure. When a subprogram is called, program locality structures will be redefined according to the localities present in the newly called subprogram. Therefore, the memory requirements of a program are also redefined by the recently called subprogram. However, when a subprogram returns, the memory requirements of the main program have to be restored. This problem is handled by creating a new activation record for each subprogram whenever it is called. The activation record contains the most recent information generated by memory directives. In particular each activation record contains the values of (P,X) pairs, where X is the memory allocation request and P is the priority of allocation. Each activation record has a pointer pointing at the previous one, thus, forming a dynamic chain connecting all activation records in the order of their creation.

The cost of executing memory directives has also been discussed in this chapter. A variation of ALLOCATE directive structure may be used to reduce the frequency of executing a directive.

The compiler directed policy can be implemented in such a way that a directive does not have to be located inside a loop structure, where it has to be executed several times.

The performance of CD in multiprogramming systems is of significance importance to this study. The CD policy is designed to be able to react to the constantly changing status of the free memory available on the system due to multiprogramming. For this purpose, CD incorporates a swapping mechanism. The swapping mechanism initiates a swapping process if the minimal memory requirement of a running process exceeds the amount of free memory available on the system. Moreover, the swapping mechanism incorporates a partial swapping strategy. Partial swapping allows a swapped out process to maintain a resident set in memory. However, the resident set size of a swapped out process is reduced to its minimal memory requirement specified by the directive associated with its lowest level locality.

The performance of CD in a multiprogramming system is evaluated in the next chapter. We will examine the fault rate characteristics of CD among other performance measures. The usefulness of partial swapping is investigated. Finally, we will compare the performance of CD with the performance of WS in a multiprogramming system.

CHAPTER 4

PERFORMANCE EVALUATION AND MEASUREMENTS

4.1. Introduction

The importance of performance and its evaluation in all technical fields is obvious. Ferrari [22] considers performance evaluation as indispensable for the viability of any technical system as the *functionality* and *economicity*. The previous chapter has addressed the other two categories: functionality and economicity of CD. The main goal of this chapter is to evaluate the performance of CD in a multiprogramming system.

The term performance is understood in the context of the performance indexes used in this study. The most common performance index of paging systems is the page fault characteristics. The number of page faults, F, is a significant index by itself whcih serves as a measure of the traffic between virtual and real memory. It also reflects the lifetime of a process: the lifetime of a process is inversely proportional to F. A process's lifetime is commonly used to model program behavior [13], [19]. Also, F can be used as a measure of a process's *virtual* turn around time. A virtual turn around time ignores the delay time in queues waiting for other processes to be served However, the virtual turn around time differs from that obtained from a uniprogramming environment. This difference results from the swapping activity which is a characteristic of multiprogramming systems only. The virtual time, VT, of a process is given by

$$VT = T + F \times L$$

where T is the length of the reference address trace; each memory reference is one time unit. F is the number of page faults generated during the execution time of a process. L is the time needed to service a page fault. L includes the time needed to interrupt CPU and to transfer control to a paging device; the seek time needed to locate a missing page in the virtual storage; and the time to transfer a page from disk (the virtual storage) to main memory. The *real* turn around time, RT, of

a process includes the waiting time in system queues:

$$RT = T + F \times L + Q$$

where Q is the time a process spends in the system's queues waiting for a service. In this study we find the number of page faults for each process in the system, F_p, and for all processes in the system, F_{sys}, where

$$F_{sys} = \sum_{p=1}^{N} F_p .$$

The space time cost, ST is another performance index, commonly used to evaluate memory management policies. ST is the time integral of the memory space occupied by a process. Obviously, a process may occupy memory space while it is running, or while in the process queue (PQ) waiting for a time slot, or while in the fault queue (FQ) waiting for a page to be paged into main memory. The *real* space time cost of a process is given by

$$ST = \sum_{i=1}^{I} S_i + L \times \sum_{f=1}^{F} S_f + \sum_{q=1}^{Q} S_q$$

where S_i is the space occupied by a process at virtual time, i; S_f is the space occupied by a process during the service of a page fault; and S_q is the memory space occupied by a process while waiting in the process queue for a CPU time quantum. Space time cost is a system performance index, rather than a process specific. From the user point of view, it is desired to minimize the running time of a process irrespective of the memory space it occupies during its execution.

Maximizing the throughput of the system is a desired goal from the system's point of view. The throughput is the number of jobs completed per unit time. With the throughput in mind as a performance index, the space time cost becomes an important criterion of performance. The results of queuing network analysis claim that a maximum throughput can always be achieved if each process in the system runs with a minimal space time cost [12], [20]. A theoretical support for this claim is based on the assumption that memory capacity is completely utilized. Assume that the total memory space is θ pages, and N processes are running for t time units. The average space time cost per process is given by:

$$ST = \frac{\theta \times t}{N} \quad .$$

The system throughput, ϕ, is given by

$$\phi = \frac{N}{t} \quad or \quad N = t \times \phi, \; hence, \; ST = \frac{\theta \times t}{\phi \times t} \quad .$$

The above formula implies that maximizing the system throughput, ϕ, can be achieved by minimizing the space time cost of every process in the system, or equivalently, minimizing the overall system space time cost. The overall system space time cost, ST_{sys}, is given by

$$ST_{sys} = \sum_{p=1}^{N} ST_p \quad .$$

The empirical results presented in this chapter contradict the above conclusion. However, the space time cost is still an important criterion of performance. Memory management policies have been designed and proposed to optimize the space time cost of a running process; among these are WS [18] and DMIN [10] (an optimal dynamic memory management policy).

The average memory size allocated to a process, or the average resident set size of a process, V, is commonly used as a performance measure of memory management policies. V is useful in studying the locality property of program behavior; it also helps evaluating the ability of a policy to measure the memory demands of a program.

We have already mentioned that throughput, ϕ, is used as a system performance index. A system manager would like to increase the output of his system by maximizing the throughput. However, this should not happen at the expense of slowing down the execution of some processes. A tradeoff must be made between the interests of individual processes and the system as a whole, e.g., minimizing a process's turn around time versus maximizing system's throughput.

A multiprogramming specific measure index is the swapping rate. Program behavior in a multiprogramming system is not a function only of its intrinsic properties, it also depends on the behavior of other processes in the system. For global memory management policies, a running process may replace the pages of any other process. For local dynamic memory management policies, a running process may swap out of memory the resident set of any other process. The swapping rate

is defined in this thesis as the total number of a process's pages swapped out of memory as a result of a swapping operation initiated by a running process. The swapping rate, Σ, is a significant performance index by itself. Moreover, Σ has an impact on the page fault rate, as we have discussed in Chapter 2. Also, Σ is responsible for the anomalous behavior of WS (see Chapter 2).

In this chapter the performance measures, discussed above, are used to evaluate the performance of CD, whcih will be compared with WS. The WS policy is chosen because it has been advocated in the literature [20] as a near optimal policy. Besides, most of the dynamic, nonglobal policies proposed to manage memory hierarchies are derivatives of WS. For example, the Damped Working Sets (DWS) [36] modifies the WS slightly to improve its behavior during interlocality transitions. DWS outperforms WS by no more than 10% in terms of minimizing the space time cost [20]. The Sampled Working Set (SWS) [34] has been proposed to reduce the implementation cost of WS. Ferrari and Yih [23] proposed the Variable Interval Sampled Working Set policy (VSWS) which combines the properties of SWS and DWS. The performance of VSWS is comparable to WS. Global policies which are not WS descendants, such as global LRU and global CLOCK, have been assumed to perform worse than WS [20]. The page fault frequency policy (PFF) [14] also achieves similar to WS performance. Carr's proposed policy WSclock [13] is an approximation and global implementation of WS; WSclock performs nearly the same as the pure WS.

It most be pointed out, however, that CD is not being compared with the optimal. But since WS and its variations are considered to have near optimal performance, CD is compared with the WS policy. This chapter presents compelling evidence that CD performs better than WS in many aspects.

Before comparing CD with WS the characteristics of CD are examined: namely, its dynamic behavior, the partial swapping mechanism employed by CD, and the impact of the context switch on CD's performance.

4.2. Modeling CD

The multiprogramming model used in Chapter 2 for evaluating the performance of WS is used for modeling CD (Figure 4-1). Following is a description of CD's implementation.

Each process is represented by its virtual address trace. A trace contains both virtual addresses and memory directives. The directives are introduced manually into the source level code. Each directive is represented by an integer number with a value larger than 1000. The most significant digit is the priority index, P, of the directive and the rest of the digits constitute the memory request, X. For example, a reference of the form 2120 is interpreted as a directive with $P=2$ and $X=120$.

Each process maintains a list of its referenced pages in the main memory. The memory space reserved for a process is determined by the X value of the last processed directive. All paging activities of a process occur within its specific memory area. The resident set of a process grows or shrinks upon processing a directive or as a result of a swapping operation.

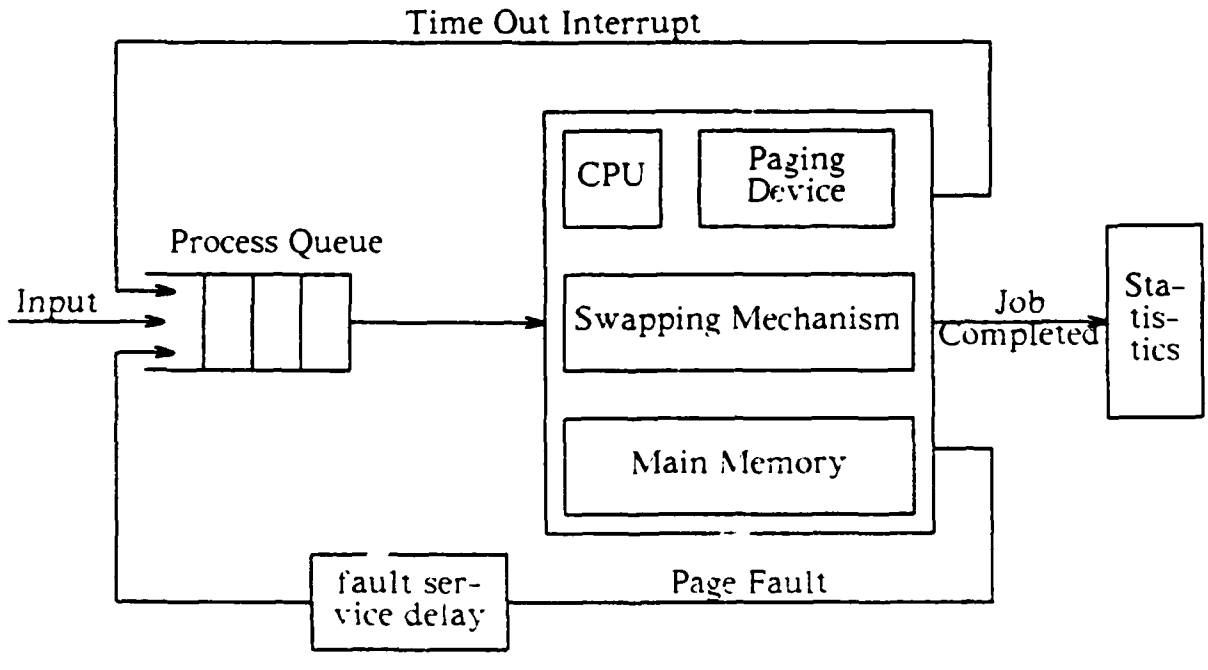


Figure 4-1: Multiprogramming model

The CD policy does not have any control parameter. For this purpose, CD exhibits no controllability problems at run time. The directives are inserted at compile time, and evaluated at run time in light of the status of the free memory. For each θ value, CD generates only one value for each of the performance indexes described in Section 1, whereas WS needs to be tuned in order to achieve a particular performance. The window size, τ, has to be properly chosen. Moreover, different values of τ might be needed to optimize different performance criteria.

System parameters are used as control parameters in order to generate more results of CD and to study the performance of CD in different environments. For example, a wide range of θ is used to demonstrate the performance of CD in small and large memory systems. The value of θ is varied from 6 to 500 pages. Also, the multiprogramming level, MPL, is varied between 3 and 10. Another system variable is the context switch, CS. Several values of CS are used ranging from 100 to =1000 time units.

The CD policy has the option of using or not using the partial swapping feature described in Chapter 3. The partial swapping mechanism is implemented as follows. Each process keeps a record of its current allocation and the memory request associated with $P=1$. The swapping mechanism keeps a circular list of all the processes in the system and a pointer pointing at the next candidate process for swapping. Upon invoking the swapping mechanism, the processes are periodically examined searching for a process occupying memory with $P>1$. If such a process is found, its memory allocation is reduced to that associated with $P=1$; this value is stored in the directive record (used to store the values of a directive's parameters) of the process. The difference in memory space, between the old X and the new X, is added to the free memory pool. If all the processes have been forced to run with $P=1$ and the free memory pool size is too small to satisfy the current memory request, a total swapping is applied, i.e., the entire resident set of a process is pre-empted.

When a process gains control of CPU, it is assigned a memory space according to the values found in its directive record. Initially, the directive record contains the value of the minimal memory space a process is entitled to have. In this model, this value is equal to 0, i.e., the resident

set of each process is initially empty. When a process is removed from the control of CPU for a time out interrupt, or for a page fault service, its directive parameters are remembered in the directive record. The CD policy does not keep a record of the members of its resident set at a time of relinquishing CPU. Upon regaining control of the CPU, CD demands its resident set's pages back into memory; WS is implemented in a similar manner.

4.3. CD Characteristics

4.3.1. Dynamic memory allocation

The amount of physical memory allocated to a program vary during execution for two reasons. The first one is attributed to a program's intrinsic locality characteristics. A transition from one locality structure to another results in a change in the memory requirement of a program. Therefore, the amount of memory allocated to a process may vary every time a new directive is executed and its request is satisfied. Variable memory allocation also occurs within a locality structure. The amount of memory allocated to a process may be reduced due to a partial swapping operation. This happens when a process is occupying memory with $P > 1$, i.e., low priority. Also, a process may switch its memory allocation from that required by a lower level locality to a larger one requested by a higher level locality; this is viable because the size of free memory may be increased if a process releases some memory or a process completes its execution.

The variation in the memory size allocated to a process is not expected, however, to be abrupt. A process is expected to spend some time inside a locality; therefore, directives are spread apart by the duration of a locality. Once a directive is processed and a particular request is satisfied, the size of free memory is not expected to change until another process executes and changes its memory requirements. For both these reasons, it is not expected to notice an abrupt change in the amount of memory allocated to a process over execution time. In this section we report some results about the dynamic memory allocation obtained under CD.

In Figure 4-2, the memory space allocated to a process is plotted versus real time. Five plots are shown in the figure, one for each of the programs. The plots are generated for MPL=3 and three values of θ (θ=50, 100, 300). Consider, for example, program MAIN. For θ=50, all memory requests larger than 50 pages are not satisfied. Memory allocation varies between 1 page and 17 pages according to the directive:

ALLOCATE (2,17) *else* (1,1)

The memory request, X=60, generated by the directive

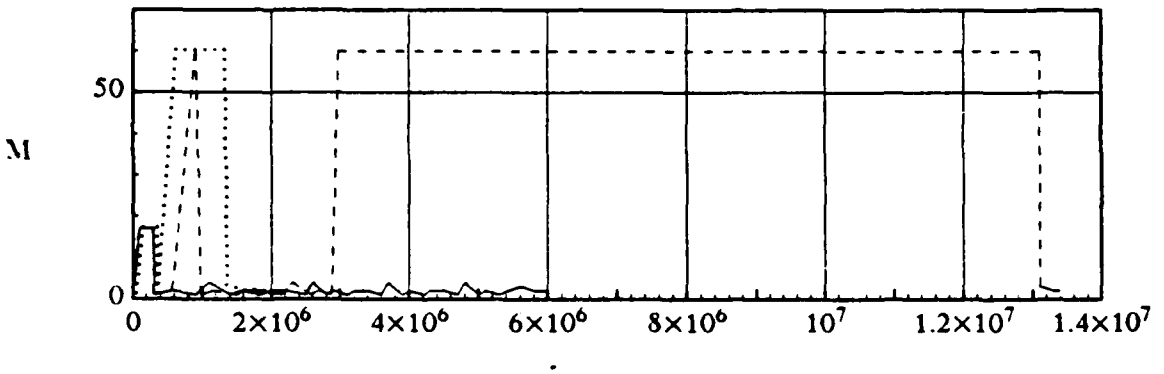
ALLOCATE (2,60) *else* (1,2)

cannot be satisfied when θ=50. However, 60 pages can be allocated when θ=100. Depending on the size of the free memory pool, memory allocation varies between 1, 2, 4, and 60 pages; this variation represents intra-locality transition. An example of transition from one locality to another (inter-locality transition), is illustrated in the time region, $t=1.28\times10^6$ and $t=1.31\times10^6$, where the amount of memory allocated to program MAIN changes from 60 to 3 pages. With larger values of θ, memory allocation within a locality structure seems to be stationary; the first request of a directive (and the largest) is allocated most of the time. For program MAIN, 60 pages are always allocated whenever a directive is executed of the form *ALLOCATE* (2,60) *else*.. Similar observations can be made from the analysis of the rest of the figures.

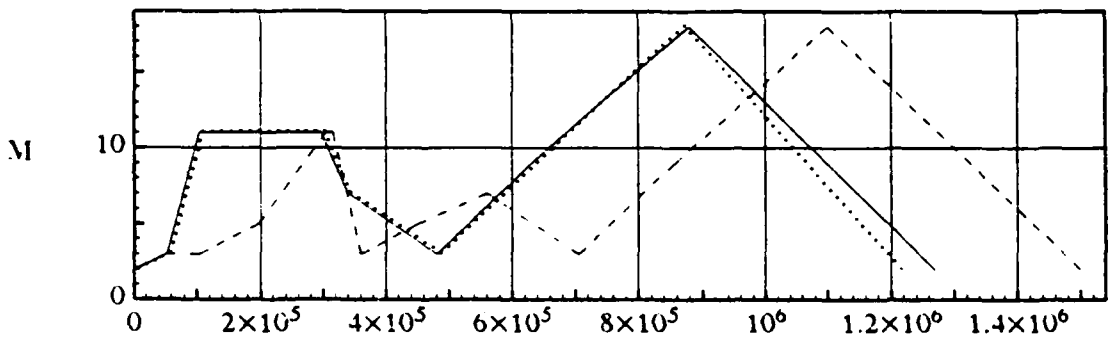
Compared to other dynamic policies, such as WS and global algorithms, CD does not exhibit high implementation costs. A program's resident set does not have to be updated or computed at every memory reference time. The number of times a resident set has to be updated is limited by the frequency of generating directive requests. The plots in Figure 4-2 show that for the five programs, memory allocation does not change in a rapid continuous fashion; it is rather discrete and widely spread over time.

4.3.2. Partial swapping

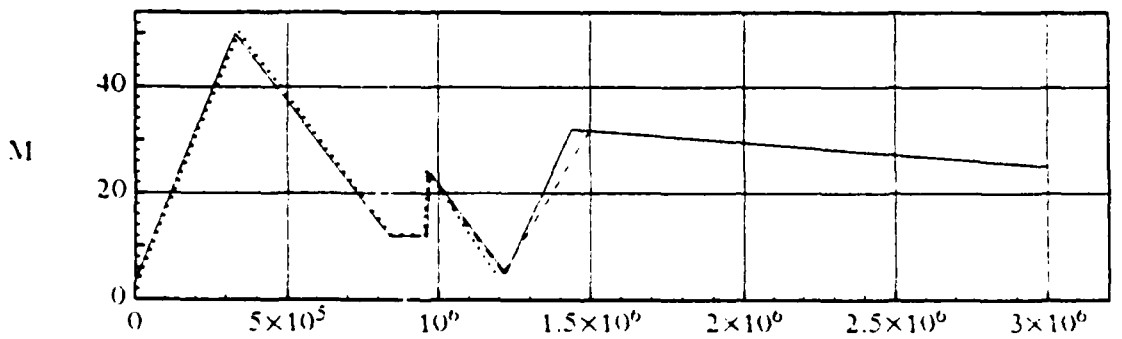
A major characteristic of CD is that it prohibits any program from running unless there is enough memory space to allocate at least one level of its current locality structure. The CD policy



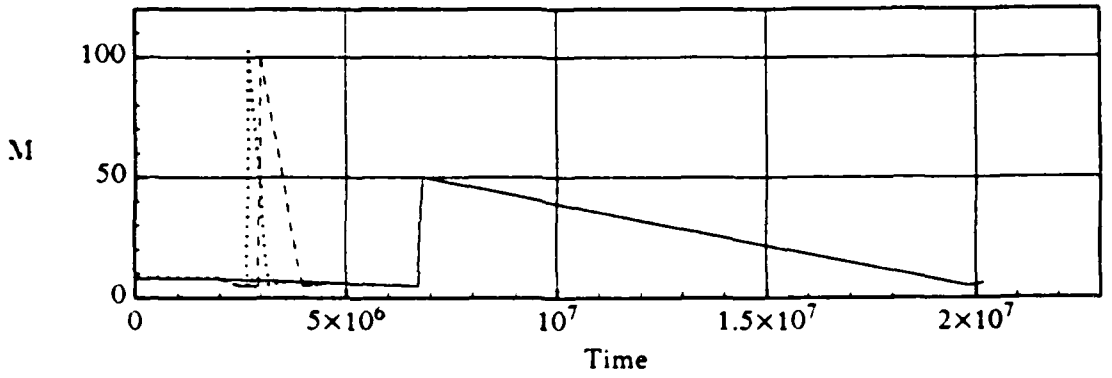
4-2a: MAIN, θ=50 ---, 100 ...., 300 - - -



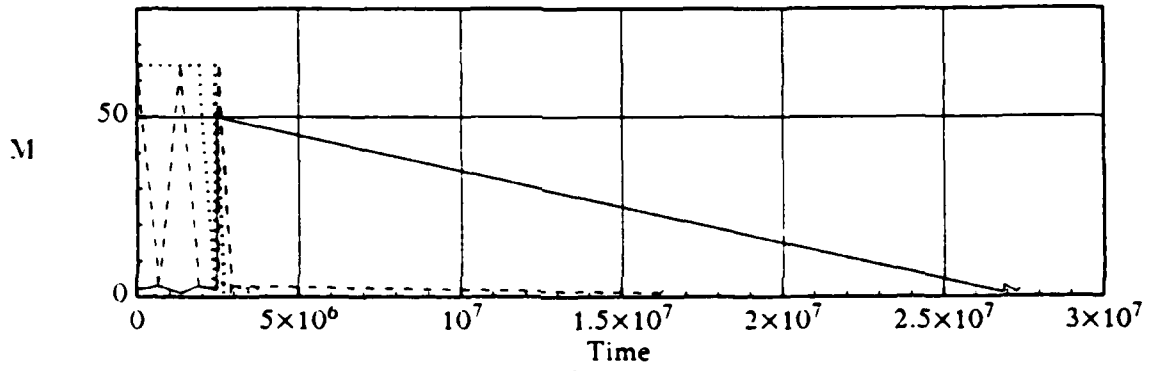
4-2b: FIELD, θ= 50 ---, 100 .... 300 - - -



4-2c: INIT, θ= 50 ---, 100 ...., 300 - - -



4-2d: CONDUCT, θ=50 ---, 100 ...., 300 - - -



4-2e: HWSCRT, θ= 50 ---, 100 ...., 300 - - -

Figure 4-2: Dynamic memory allocation under CD

incorporates a swapping mechanism to facilitate this feature. The swapping mechanism has been discussed in Chapter 3. Partial swapping is introduced in order to give a process a chance to keep some pages in memory before it is completely demoted. Hopefully, with partial swapping more processes can share the memory and CPU resources at no risk of thrashing. One may argue that partial swapping may increase the number of processes in the system at the expense of generating more page faults. In a multiprogramming environment, it is not easy to agree or disagree with this argument on a purely theoretical basis. On one hand it might be true that less memory allocation results in more fault rate, or at least should. On the other hand, the lowest level locality might have a time duration even longer than the context switch time interval given to a process. In this case partial swapping can have only a positive impact on the performance of an individual process

and of the system. Besides, total swapping may result in extra completely unutilized memory space.

The empirical results reported in this section demonstrate the impact of using a partial swapping strategy along with a total swapping. For MPL=10 and θ=50, the total number of system page faults is 39015 with partial swapping turned on; without partial swapping the page faults increased to 53220, a difference of 12105 faults. However, for larger values of θ the big difference disappears. For example, the number of page faults for θ=150 with partial swapping on is only 3 faults less than the fault rate without partial swapping. For θ=200, 300, and 400 the difference disappears completely.

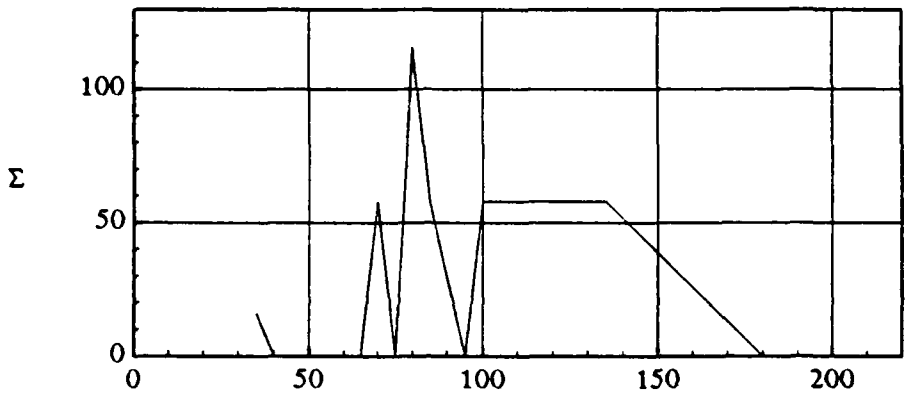
For each process in the system the total number of pages, Σ, swapped out from the resident set of a process is recorded. Figure 4-3 presents a plot of Σ versus θ for programs MAIN, INIT, and HWSCRT with MPL=5.

Our results show that partial swapping can indeed result in more page faults. However, for high memory contention cases, partial swapping has the tendency to generate less faults than total swapping.

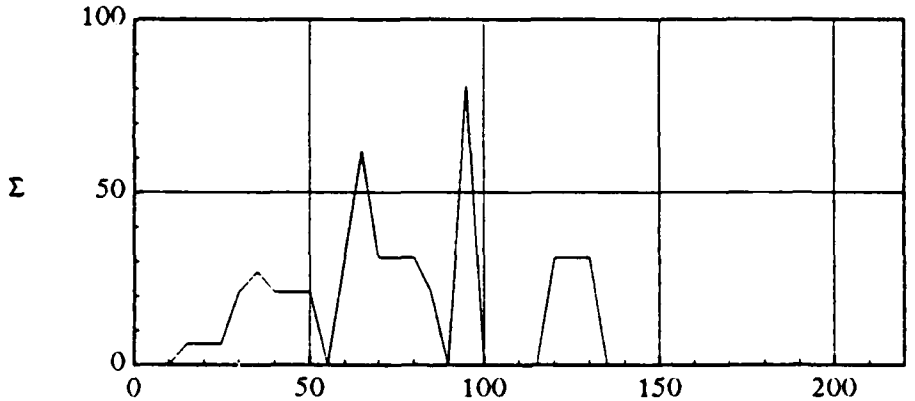
4.3.3. Effect of context switch

One reason for a process to relinquish CPU is to use up its context switch interval where a time out interrupt is generated. A process may actually use all of its time slot if it is not interrupted during the context switch period. In our model, the only other interrupt is caused by a page fault. Therefore, if the average lifetime of a process, i.e., the time between successive page faults, is larger than the context switch value, CS, a process is likely to use up its time slot before a page fault occurs. Similarly, if the time between successive faults is shorter than CS, a process is likely to lose control of CPU before its context time runs out.

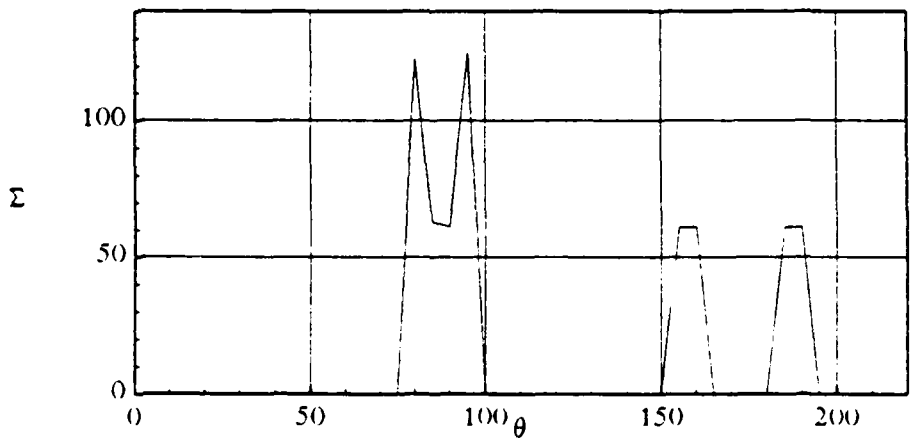
The average lifetime of a process is inversely proportional to the fault rate. The average virtual lifetime, G, of a process is defined as $G = T/F$ where T is the length of the address reference



4-3a: MAIN, MPL=5



4-3b: INIT, MPL=5



4-3c: HWSCRT, MPL=5

Figure 4-3: Partial swapping rate versus θ

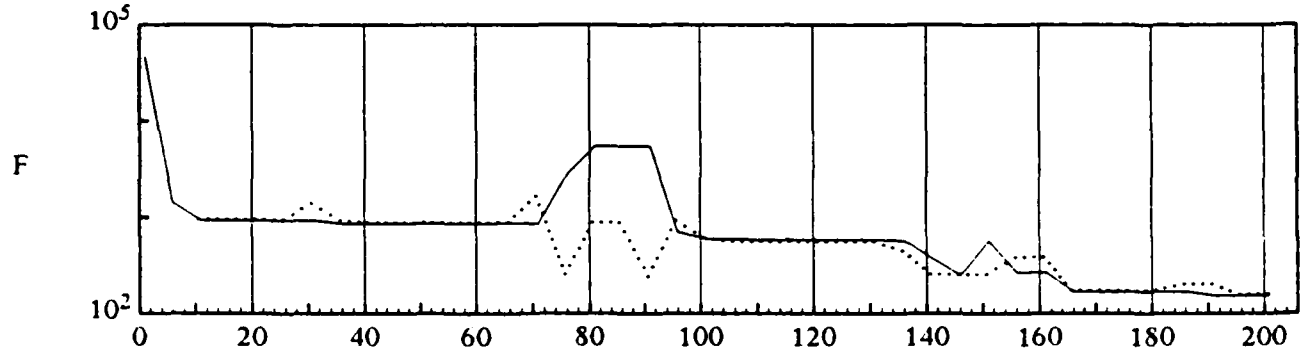
string, and F is the number of page faults. The average virtual lifetime is maximum if F is minimum. The minimal number of faults in a demand paging system is equal to the number of pages in the virtual space of a process. The maximum value of G for the programs used in our experiments is given in Table 4-1. Averaging over all of the programs, the maximum average lifetime is 365. In reality the lifetime of each process is lower than the values given in Table 4-1 because the actual number of faults is much higher than the absolute minimum.

All of the results reported in this thesis use CS=1000. This value is large enough to exclude the impact of CS on the results. Mainly we are interested in paging related characteristics of program behavior. However, we report in this section more results using smaller values of CS. The fault rate characteristics are observed under different values of CS.

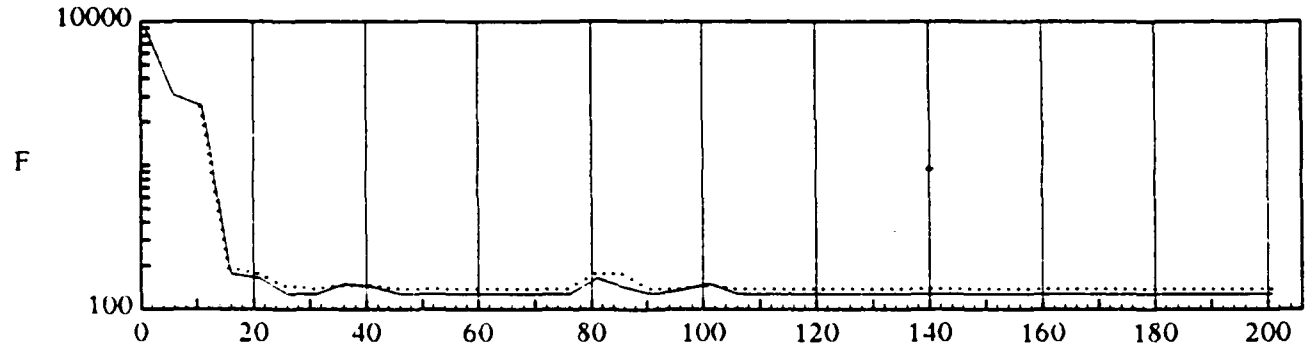
In Figure 4-4 we plot the fault rate versus θ for each program (MPL=5). Two values of CS are used, CS=100 (solid line) and CS=1000 (dotted). From the curves in Figure 4-4 we note that most of the programs favor larger values of CS. Program MAIN, with G_{max}=1017, generates significantly less faults, with CS=1000, than with CS=100, especially with θ in the range 75–100 pages. For example, $F(\theta=80,CS=100)=5524$ and $F(\theta=80,CS=1000)=852$, a difference of 4872 faults. For large θ values, the difference page faults for different CS values disappear since, for large θ values, the number of faults is considerably low, and a process is allowed to use all of its time slice. Moreover, the swapping rate is considerably lower with large values of θ, and therefore, a process is likely to retain its resident set pages when it regains control of CPU during the next context switch.

Table 4-1
Maximum lifetimes of programs

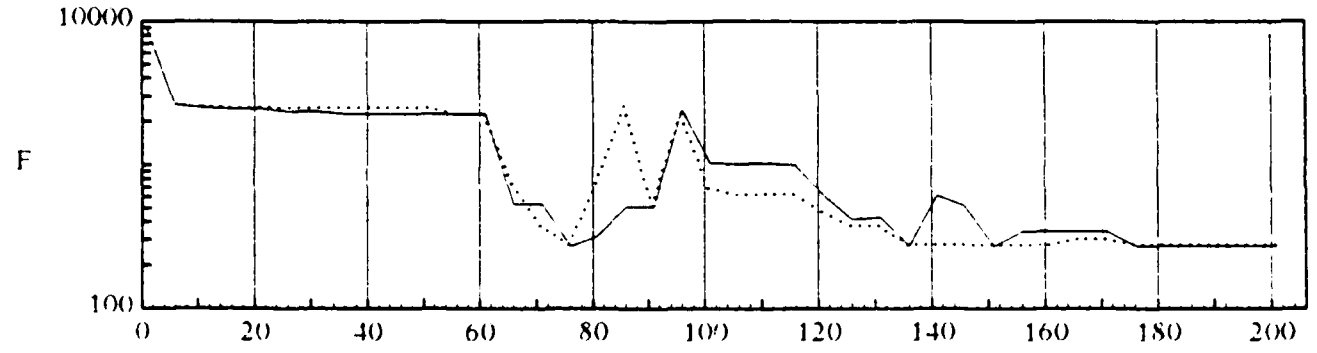
Program	Ref. Length (T)	Virtual Size	G_{max}
MAIN	79,325	78	1017
FIELD	10,823	60	181
INIT	10,745	174	62
CONDUCT	82,452	291	283
HWSCRT	22,721	76	299



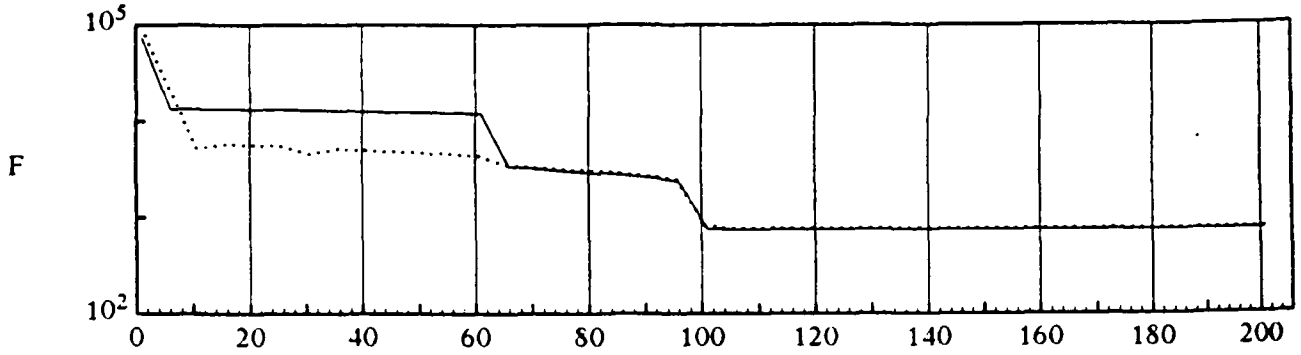
4-4a: INIT, MPL=5, CS=100 ---, 1000 ...



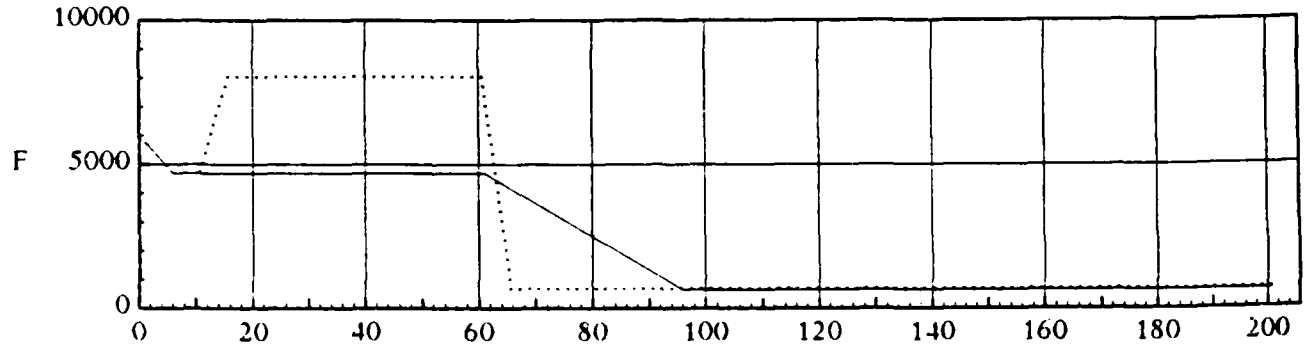
4-4b: FIELD, MPL=5, CS= 100 ---, 1000 ...



4-4c: INIT, MPL=5, CS=100 ---, 1000 ...



4-4d: CONDUCT, MPL=5, CS=100 ---, 1000 ...

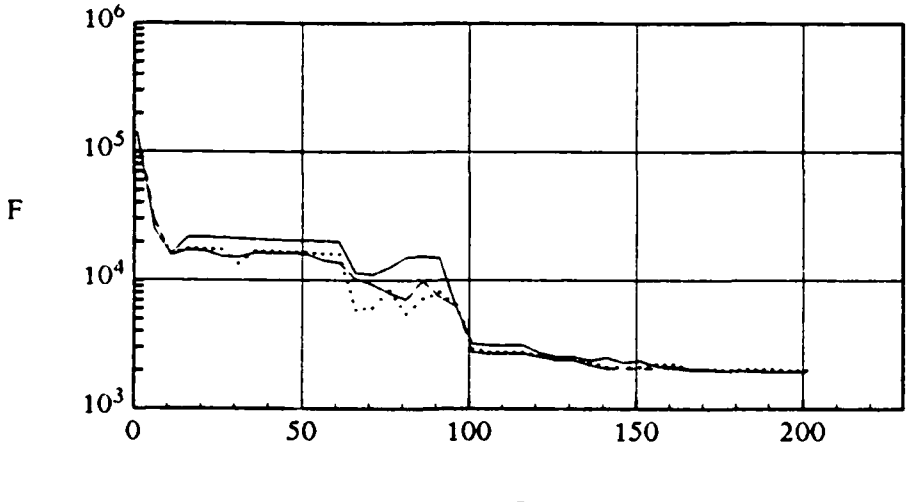


4-4e: HWSCRT, MPL=5, CS=100 ---, 1000 ...

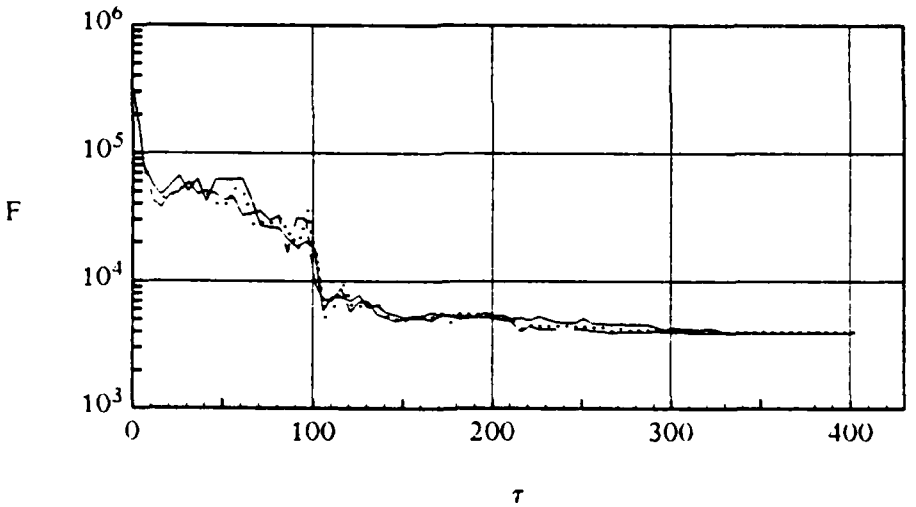
Figure 4-4: Effect of context switch on page faults

System fault's curves for three values of CS (100 solid, 1000 dotted, 2000 dashed) are shown in Figures 4-5a and 4-5b for MPL=5 and 10, respectively. The curves are almost identical for $\theta > 100$ and $\theta > 300$ for MPL=5 and MPL=10, respectively. For smaller θ values, smaller CS values generate a larger number of faults. For small θ values, the swapping activity is considerable and, therefore, it is possible that a process be swapped out before its next time slot. Using a relatively large CS allows a process to benefit from those pages it has paged into its resident set.

However, using a large CS value affects the response time because a process has to wait too long in the process queue before its next scheduling time. Small CS values, as discussed above, have the tendency to generate more faults and consequently increase the turn around time of a process. There is a tradeoff between response time and turn around time. Response time, however, has to be



4-5a: SYSTEM, MPL=5, CS=100 ---, 1000 ...., 2000 - - -



4-5b: SYSTEM, MPL=10, CS=100 ---, 1000 ...., 2000 - - -

Figure 4-5: Effect of context switch on page faults

acceptable to human norms. And therefore, a maximum response time can be enforced by using a global context switch, g. The distribution of g among the processes depends on the lifetime of a process and the number of processes in the system. The general criterion is that a process should be allowed to continue using CPU as long as it does not generate a reference to a non-resident page, i.e., page fault. However, the smooth behavior of a process should not be a reason to keep other processes waiting in the queue; after all, these processes may have a smooth behavior as well.

Therefore, a process should be pre-empted from CPU if it exceeds a threshold value. Following is a dynamic strategy for allocating time quantums to running processes.

Let g be a global context switch; g is set to a maximum value m. Also, let N be the number of processes which have not been scheduled yet to run during one scheduling cycle; a scheduling cycle is completed when all the processes in the system have used CPU once for some time. Define a threshold, h, as $h = \frac{g}{N}$; g is always evenly distributed among the remaining processes in the system. Every time a process leaves CPU after some time t, g is updated as $g = g - t$; and N is updated as $N = N - 1$. The time a process spent using CPU is determined by an interrupt due to a page fault or a time out interrupt after h time units, whichever occurs first. We further illustrate this strategy using an example.

Example 4-1:

Assume that there are 4 processes in the system. Let $m = 1000$ time units; i.e., the maximum response time for any process is 1000. Initially, $g = 1000$ and $h = 1000/4 = 200$. Let process P_1 run until a fault occurs after 100 time units, $t = 100$; i.e., P_1 does not use all of the time it is entitled to ($h = 200$). At this point g is updated as $g = 1000 - 100 = 900$; and g is distributed among three processes since $N = 4 - 1 = 3$ ($h = 900/3 = 300$). Note at this point that the remaining processes in the system have a higher threshold than did P_1 when it controlled CPU. Next P_2 runs and uses up all of its time quantum (300 units) before it generates a fault. All parameters are updated as $g = 900 - 300 = 600$; $N = 3 - 1 = 2$; and $h = 600/2 = 300$. Assume that P_3 executes until a page fault occurs after 150 time units. The value of g now becomes $g = 600 - 150 = 450$; $N = 2 - 1 = 1$, and $h = 450$. Process P_4 can use CPU for 450 time units unless it generates a page fault ; assume that a fault occurs after 400 time units. Now g is reset to 1000 and a new cycle begins. Note that no process in the system may wait in the queue more than 800 time units, and each process is allocated at least 250 time units.

The above scheme allows smoothly behaving processes (with low fault rate) to take advantage of the short lifetime of heavily faulting processes. At the same time heavily faulting processes

are not punished for bad behavior: a heavily faulting process is scheduled to run after at most m time units. In the above example, P_2 could use CPU for 300 time units because P_1 did not use all of its time. Similarly, P_4 could use CPU for 400 time units because P_3 was pre-empted before its time had expired. However, P_1 is rescheduled after 1000 time units from the time it first controlled CPU. Using static CS distribution, P_1 could have been scheduled after 650 time units. Of course this is a shorter response time, but it makes little difference if m is chosen within the range of human acceptable reaction (few milliseconds for example) for interactive systems. Moreover, two processes could be interrupted (P_2 and P_4) although they could have used CPU for useful work.

The notion of response time is, mostly, applicable to interactive systems. In batch processing systems, response time has little significance. Therefore, for batched scheduled jobs, it is more effective if a process is allowed to execute until it generates a page fault.

Dynamic time allocation is still to be further investigated. One way to pursue this issue is to look into the possibility of using memory directives introduced in this thesis, or possibly some *time* directives, to guide a dynamic time allocation strategy. In this thesis we investigated only static time allocation.

4.4. CD Versus WS

Simulation is performed for several values of θ ranging from θ = 6 to 200 pages. Small values of θ represent the case of high memory contention characterized by a relatively high rate of swapping. Larger values of θ are used to evaluate the performance of CD and WS when there is enough memory to allocate the resident sets of programs as requested by CD or defined by τ, the WS parameter. Four levels of multiprogramming (MPL) are used: MPL=3, 4, 5 and 10. MPL=10 is achieved by running two copies of each program simultaneously. For MPL=3 high memory contention results for θ < 30 pages. For MPL=10, memory contention is observed for θ < 150 pages.

Next CD is compared with WS in terms of the page faults, the space time cost, the system throughput, and controllability.

4.4.1. Page faults

Minimizing the turn around time of a job is a primary performance objective from the user's point of view. In a virtual memory system, this objective can be achieved by minimizing the page faults of a user's process. However, minimizing the faults of a process in the system may adversely affect other running processes' page faults and worsen the overall system performance. In the next subsection we study the page fault characteristics when the objective is to minimize the faults of individual processes.

4.4.1.1. Page faults of individual processes

In a uniprogrammed system, WS can be easily tuned to achieve the absolute minimum number of page faults by choosing a relatively large value for τ. Earlier experiments [20], [3] have always assumed a uniprogrammed system with infinite memory where τ value is not restricted by the memory size. However, in practice τ is restricted by the finite memory capacity available on the system. In a multiprogramming system the page faults of a process is affected by other processes running in the system; therefore, large τ values may not always generate a low number of faults. In Chapter 2, t was shown that increasing τ may result in increasing the number of faults, i.e., anomalous behavior. Also, larger τ values yield large working set sizes which lead to a memory contention problem among processes in the system.

Table 4-2a (MAIN)
CD compared with the minimal achievable page faults under WS
with corresponding space time costs

	Page Faults			ST Cost(10^6)			τ
θ	CD	WS	Δ_F%	CD	WS	Δ_{ST}%	WS
6	923	1743	89%	3.84	9.07	136%	196
10	923	978	06%	3.84	5.77	50%	6
20	872	921	06%	3.79	5.71	51%	6
25	855	921	08%	4.33	5.71	32%	6
50	855	380	-44%	4.33	22.7	424%	6900
100	169	302	79%	5.22	24.7	373%	9900

Table 4-2b (FIELD)
CD compared with minimal achievable page faults under WS
with corresponding space time costs

	Page Faults			ST Cost(10^6)			τ
θ	CD	WS	Δ_F %	CD	WS	Δ_{ST} %	WS
6	173	7903	4468	2.858	31.6	1005	1
7	173	3795	2094	2.858	33.1	1058	6
8	173	3172	1733	2.858	27.2	862	386-436
9	173	2892	1572	2.858	26.2	816	441-∞
10	173	3357	1840	2.858	34.1	1093	6
11	173	2784	1509	2.858	30.8	978	381
12	136	1307	861	2.899	11.9	310	261
13	136	1217	795	2.899	12.3	324	396-436
14	136	1153	748	2.899	12.1	317	771-1101
15	136	1163	755	2.899	12.3	324	221-226
							451-511
16	136	1146	743	2.899	13.1	352	386
17	136	1101	710	2.899	13.5	366	581-761
18	136	341	151	2.899	3.91	35	261
20	136	219	61	2.899	3.1	7	381-396
25	143	149	4	2.768	2.98	8	1301-1501
30	136	134	00	2.899	2.62	-9	1601-
35	136	104	-23	2.899	3.03	5	1801-2301
40	136	113	-17	2.899	2.92	1	4201-5401
45	136	107	-21	2.899	3.04	5	921-961
50	136	106	-22	2.899	3.2	10	6501-
100	136	69	-49	2.899	3.68	27	4701-6001

For CD, the number of faults is a function of θ only ($F_{CD}(\theta)$), although θ is not a control parameter. In this study we use a wide range of θ values to demonstrate the ability of each policy to function in small and large memories. For each θ value CD generates one set of results including the number of faults for each process and for the system.

The WS policy is controlled by τ, the window size. Each performance index is a function of τ. For each θ and each τ, WS generates one set of results. Since we use several values of τ, several sets of results are obtained. The minimum τ value used is τ=1. An increment of 5 is used up to a value of τ=1000. A small increment is necessary to capture the behavior of WS in transitional

periods. In numerical programs, changes in locality structures occur in abrupt fashion; this is obvious from the lifetime of individual numerical programs (see reference [8]). A larger increment is used for $\tau > 1000$. The WS window size is increased until the working set size of any process exceeds the amount of physical memory, θ, where an *overload* condition is raised; in this case the results are generated for all preceding τ values and the simulation is terminated. Simulation may be continued only with larger θ values.

Each τ value is used by all processes in the system (fully detuned policy [20]). Alternatively, one can use for each process in the system a separate τ which optimizes the performance of the particular process (fully tuned policy [20]). The high overhead associated with fully tuned policy res-

Table 4-2c (INIT)
CD compared with the minimal achievable page faults under WS
with corresponding space time costs

	Page Faults			ST Cost(10^6)			τ
θ	CD	WS	Δ_F %	CD	WS	Δ_{ST} %	WS
6	2520	3686	46	13.8	24.5	78	196-376
7	2520	3150	25	13.8	24.3	765	256
8	2520	3038	21	13.8	31.7	130	6
9	2520	2610	04	13.8	28.4	106	6
10	2520	2556	02	13.8	28.7	108	6
11	2520	2525	00	13.8	28.9	109	6
12	2457	2525	03	13.19	29.0	120	6
13	2457	2519	03	13.19	43.2	228	11
14	2457	2515	02	13.19	44.3	236	11
15	2457	2511	02	13.19	45.6	246	11
16	2457	2513	02	13.19	46.0	249	11
17	2457	2514	02	13.19	50.0	279	11-16
18	2457	2514	02	13.19	60.2	356	16
20	2457	2509	02	13.19	46.5	253	11
25	945	2509	164	5.16	50.0	8695	11-16
30	945	978	03	5.16	35.6	590	81
35	945	960	02	13.41	33.2	148	66-86
40	945	947	00	13.41	48.7	263	121
45	945	947	00	13.41	47.2	252	116
50	369	947	157	11.22	61.0	444	156-161
100	273	175	-36	15.47	14.2	-8	516-1101

tricts its use. To achieve a performance close to that of fully tuned WS with relatively low overhead we find for each process τ_{F-min} which produces the minimal number of page faults ($F_{\min}(\theta,\tau)$); recall that our objective is to minimize the turn around time of individual processes. The side effects of operating with τ_{F-min} are measured by evaluating the corresponding space time costs $ST(\tau_{F-min})$ and the average working set size $W(\tau_{F-min})$.

For MPL=3, the results are shown in Tables 4-2a, 2b, 2c, for programs MAIN, FIELD, and INIT, respectively. In the first column of each table is the memory size, θ. The number of page faults generated under CD and WS are given in the next columns; for WS the number of faults is the minimal value selected from several values generated under different τ values. The relative difference between F_{CD} and F_{WS} is given by

$$\Delta_F = \frac{F_{WS}-F_{CD}}{F_{CD}}\times 100\% \quad . \tag{4-1}$$

Positive Δ_F values indicate that the number of faults under WS is larger than that under CD. For the same θ, the space time costs under CD and WS are given in the next two columns. For CD this is the only value. For WS this is the space time cost achieved using τ_{F-min}. The relative difference between ST_{WS} and ST_{CD} is given by

$$\Delta_{ST} = \frac{ST_{WS}-ST_{CD}}{ST_{CD}}\times 100\% \quad . \tag{4-2}$$

The last column shows the optimal τ for each process.

The analysis of Tables 4-2 shows that CD performs better than WS in high memory contention cases (small θ values); high memory contention is characterized by high swapping activity. Consider, for example, $\theta = 8$. The minimal faults under WS for programs MAIN, FIELD, and INIT are higher than those achieved under CD by 53%, 1733%, and 21%, respectively. Under CD, 4 swapping operations are performed to pre-empt 18 pages of memory, whereas under WS, more than 40 swapping operations are initiated. The performance of WS improves when the memory available on the system is relatively large. For example, WS produces 36% and 49% less faults than CD for θ=100 for INIT and FIELD, respectively. CD still outperforms WS for program MAIN by

Table 4-3a (MAIN)
CD compared with the minimal achievable page faults under WS
with corresponding space time costs (MPL=4,5,10)

		Page Faults			ST Cost(10^6)			τ
MPL	θ	CD	WS	Δ_F %	CD	WS	Δ_{ST} %	WS
4	10	923	1469	59%	3.841	7.59	98%	9
	20	923	947	3%	3.841	5.87	53	10
	25	923	921	00	3.841	5.71	49%	6
	30	889	921	04%	4.367	5.81	33%	10
	40	855	921	08%	4.331	5.83	35%	26
	50	855	921	08%	4.331	5.83	35%	50
	100	169	919	444%	5.221	15.6	199%	415
	150	152	258	70%	7.117	23.4	229%	6600
	200	152	79	-48%	7.117	10.8	52%	16,500-
5	50	855	1157	35%	4.331	11.9	175%	51
	100	237	921	288%	4.72	9.92	110%	50-250
	150	169	310	83%	5.21	22.9	340%	6000
	200	152	139	-9%	7.117	14.4	102%	20.000-
10	50	895	1029	20%	4.334	6.11	41%	11
	100	237	956	303%	4.72	5.96	26%	51
	150	245	564	130%	8.113	22.2	174%	6900
	200	152	474	212%	7.117	23.1	225%	7900

79%. For θ=100, the swapping rate is 0 under both CD and WS.

The improvement of WS with a relatively large memory size (θ=100 for MPL=3) is expected since the working set size of a program can grow with less restriction. Using large values of θ may result in a situation similar to a uniprogramming system with infinite memory, where WS can achieve the absolute minimal number of faults by using a relatively large τ. In a multiprogramming system it is always possible to transfer the system into high memory contention state by increasing the number of processes competing for memory space and CPU time, i.e., increasing MPL. Comparing CD and WS for small θ values can be a useful measure of the optimal MPL supported by both policies. Consider, for example, the performance of CD and WS for θ=50. For MPL=3, WS generates less faults than CD does for programs MAIN and FIELD by 44% and 22%, respectively. When the multiprogramming level is increased to MPL=4, WS generates more faults

Table 4-3b (FIELD)
CD compared with minimal achievable page faults under WS
with corresponding space time costs (MPL=4,5,10)

		Page Faults			ST Cost(10^6)			τ
MPL	θ	CD	WS	Δ_F %	CD	WS	Δ_{ST} %	WS
4	10	2501	3909	56%	8.946	35.5	275%	9
	20	173	1806	944%	2.858	21.7	659%	15
	25	136	241	77%	2.899	3.14	08%	21
	30	136	193	42%	2.751	2.81	02%	30
	40	136	161	18%	2.899	3.33	15%	31
	50	136	161	18%	2.899	3.55	22%	45
	100	136	133	00	2.899	3.76	30%	415
	150	136	64	-53%	2.899	4.25	47%	6600
	200	136	61	-55%	2.899	4.09	41%	10,500-
5	50	136	2762	1931%	2.899	38.1	1214%	101
	100	136	109	-20%	2.899	3.67	27%	951
	150	136	68	-50%	2.899	3.8	31%	5500
	200	136	61	-55%	2.899	4.09	41%	15,000-
10	50	135	405	200%	2.762	4.82	43%	21
	100	165	164	00	2.852	3.07	08%	701
	150	128	114	-11%	2.893	3.58	24%	7000-
	200	128	83	-35%	2.893	3.03	05%	1800

than CD by 8% and 18% for MAIN and FIELD, respectively. Increasing MPL further to MPL=5 and 10, the number of faults under WS exceeds that under CD by 35% and 20% for MAIN, and by 1931% and 200% for FIELD, respectively. The results for MPL=4, 5, and 10 are reported in Tables 4-3, one table for each program.

From Tables 3 we note that CD benefits from increasing MPL for the same θ value, whereas the performance of WS degrades with increasing MPL. Consider, for example, program HWSCRT (Table 4-3-e). Doubling MPL has almost no effect on the performance of CD, whereas the page faults under WS increased more than 12, 3, and 2 times for θ=100, 150, and 200, respectively.

The low number of page faults under WS, generated with larger θ values, is almost always associated with a space time cost (ST) larger than CD's. In other words, WS generates less faults on the expense of occupying more memory space for a longer time. Consider, for example, Table

4-3a for program MAIN, MPL=3, and θ=50. The WS policy generates 44% less faults than does CD (Δ_F =−44%) . However, the space time cost under WS is 4.24 times more than that under CD (Δ_{ST} =424%). For program CONDUCT in Table 4-3d, WS's improvement over CD in terms of page faults for MPL=5 and θ=100,150,200 is accompanied by excess space time cost of 25%, 186%, and 247%, respectively. On the other hand, ST_{CD} is lower for most of the time than ST_{WS} even when CD generates fewer faults than WS. For θ = 9, in Tables 4-3a, 3b, 3c, CD generates less faults than WS by 16%, 1572% and 4% for MAIN, FIELD and INIT, respectively. For the same θ, CD outperforms WS in terms of ST by 59%, 816% and 106% for the same programs.

The analysis of Tables 2 and 3 show that CD achieves better performance than WS in a small memory environment. The WS is a better policy when using a large memory size. However, for the same memory size, CD can support higher multiprogramming levels. CD is designed to respond to

Table 4-3c (INIT)
CD compared with the minimal achievable page faults under WS
with corresponding space time costs (MPL=4, 5, 10)

		Page Faults			ST Cost(10^6)			τ
MPL	θ	CD	WS	Δ_F %	CD	WS	Δ_{ST} %	WS
4	10	2520	3298	31%	13.8	22.5	63%	9
	20	2525	2544	00	13.04	42.3	224%	10
	25	2525	2521	00	13.04	29.0	123%	6
	30	2525	1113	-55%	14.05	18.9	35%	30
	40	2525	997	-61%	14.05	15.5	11%	26
	50	729	977	34%	14.05	26.9	92%	60
	100	297	274	-07%	14.6	14.5	00	415
	150	273	175	-36%	15.47	19.0	23%	2400
	200	273	175	-36%	15.47	28.7	86%	6500
5	50	729	2818	287%	5.045	38.8	669%	101
	100	273	187	-32%	9.28	9.69	04%	551
	150	273	175	-36%	9.28	13.8	49%	1000
	200	273	175	-36%	9.28	13.4	44%	950
10	50	535	1500	180%	11.7	22.8	190%	201
	100	273	523	92%	10.7	15.9	52%	600
	150	273	215	-21%	3.15	12.6	302%	2000
	200	273	175	-36%	9.3	12.8	31%	1600

changes in the memory status in a multiprogramming system. Both the hierarchical structure of memory directives and the partial swapping mechanism enhance the performance of CD.

In the above analysis we have assumed that each process can use its own optimal τ (fully tuned policy). The high overhead associated with this policy restricts its usage in real systems. Choosing one τ among the optimal ones (p% detuned policy [20]) may degrade the overall system performance. Moreover, an optimal τ for one process may not be usable by other processes. For example, the optimal τ for program MAIN (θ=45) is 6200. This τ cannot be used by INIT since the working set size (a function of τ) exceeds the available memory on the system: $V(\tau=6200,\theta=45)$ = 69 pages > θ=45. In the next subsection we consider optimizing the overall system page fault performance.

Table 4-3d (CONDUCT)
CD compared with the minimal achievable page faults under WS
with corresponding space time costs (MPL=4, 5, 10)

		Page Faults			ST Cost(10^6)			τ
MPL	θ	CD	WS	Δ_F %	CD	WS	Δ_{ST} %	WS
4	10	5043	23005	356%	96.96	268.0	176%	9
	20	4788	5507	15%	173.9	92.8	-47%	15
	25	4634	5148	11%	207.2	106.0	-9%	21
	30	4456	4952	11%	236.0	185.0	-22%	60
	40	4125	4876	18%	284.9	203.0	-29%	66
	50	3873	4637	20%	328.4	278.0	-15%	116
	100	789	2903	268%	301.7	421.0	40%	415
	150	748	572	-24%	22.22	70.6	218%	6600
	200	748	406	-46%	22.22	79.7	259%	20,000
5	50	4562	5979	31%	96.9	160.0	65%	101
	100	789	754	-04%	30.17	37.7	25%	601
	150	748	582	-22%	22.22	63.5	186%	6500
	200	748	403	-46%	22.22	77.1	247%	50,000
10	50	3873	5864	51%	38.0	105.0	176%	21
	100	789	4875	518%	30.17	202.0	570%	451
	150	748	1241	66%	22.22	84.8	282%	24,500
	200	748	677	-09%	22.22	35.3	59%	1000

Table 4-3e (HWSCRT)
CD compared with minimal achievable page faults under WS
with corresponding space time costs (MPL=5, 10)

		Page Faults			ST Cost(10^6)			τ
MPL	θ	CD	WS	Δ_F%	CD	WS	Δ_{ST}%	WS
5	50	649	4744	631%	11.33	84.4	645%	101
	100	646	378	-42%	11.33	19.5	72%	401
	150	646	155	-76%	11.33	13.3	17%	6500
	200	646	123	-81%	11.33	9.43	-16%	10.000
10	50	4680	4684	00	11.33	82.0	632%	71
	100	649	4580	606%	11.33	188.0	1559%	651
	150	646	474	-27%	11.33	23.2	105%	551
	200	646	340	-47%	11.33	17.2	52%	551

4.4.1.2. Overall system page faults

For WS we find one global τ which minimizes the overall system page faults. We then use this τ to find the corresponding page faults and space time costs of the individual processes. The results for MPL=3 are reported in Table 4-4. In Table 4-4 we compare the minimal overall system and the corresponding individual processes' page faults under WS with page faults achieved under CD. The space time costs of generating the given fault rate performance are also compared. From Table 4-4 it is easy to see that CD produces less faults than WS, irrespective of the maximum memory available on the system. However, the performance of CD is much better than that of WS when the memory contention is very high. For θ = 6, WS generates 164% more faults than CD does. For θ = 25 CD still outperforms WS by 85%. CD also outperforms WS on the individual processes level. For θ = 50, WS generates 5%, 21%, 157% and 50% more faults than CD does for programs MAIN, FIELD, INIT and the overall system, respectively.

The results for MPL=4,5 and 10 are reported in Table 4-5. For MPL=4, CD outperforms WS for θ<150. The improvement is higher for smaller θ values, e.g., 188% for θ=10. Similarly, for MPL=5, CD outperforms WS for θ<150; Δ_F =8% for θ=50 and 100. For MPL=10, WS generates 73%, 333%, and 38% more faults than CD for θ=50, 100, and 150, respectively. Note that when

Table 4-4
Optimizing system performance. MPL=3

	Δ_F %				Δ_{ST} %			
θ	MAIN	FIELD	INIT	System	MAIN	FIELD	INIT	System
6	85	2464	35	164	113	733	136	215
7	65	2094	23	133	96	1058	123	248
8	45	1956	21	119	83	1048	130	248
9	16	1864	4	95	59	1072	106	232
10	6	1840	1	91	50	1093	108	235
11	5	1836	00	89	48	1090	109	235
12	24	2016	3	89	66	1356	120	362
13	28	2110	3	91	72	1345	228	361
14	17	2041	2	85	60	1363	236	367
15	5	1930	2	77	53	1407	246	328
16	4	1921	2	77	52	1407	249	380
17	20	1163	2	52	64	869	340	365
18	6	1151	2	48	54	1114	356	408
20	7	1136	2	47	54	1176	363	384
25	9	1076	166	85	33	1269	1111	762
30	8	18	6	7	35	11	200	98
35	8	18	6	7	35	14	16	31
40	9	29	1	6	53	20	138	104
45	8	18	2	5	52	24	124	95
50	8	21	157	50	54	19	288	191
100	131	-48	-27	15	377	18	31	106

MPL is doubled (from 5 to 10) the improvement of CD over WS increases. The CD policy outperforms WS for MPL=5 and θ=50 by only 8%; however, a 73% improvement is achieved for MPL=10, as well as for MPL=5 and θ=150. WS generates less faults than CD by 36%; for the same θ value (150) and MPL=10 the number of faults under CD is increased from 2048 to 4515, while the page faults under WS increased from 1303 to 6241, i.e., CD's faults increased by 2.2 times and WS's faults by 4.8 times. The outcome is a CD improvement of 38% over WS. For θ=200, the WS's improvement over CD decreased from 49% for MPL=5 to less than 2% for MPL=10.

As has been concluded from the analysis of individual processes, CD is more capable than WS for supporting higher MPL for the same memory size. Recall that CD forces every process in the system to run with minimal memory allocation in high memory contention cases. A process

running with a priority index $P > 1$ for some MPL, could be forced to run with $P = 1$ (less allocation) for a higher MPL.

The second major column in Table 4-5 shows the excess space time cost that WS produces over CD for the overall system and the individual processes. The very large ST exhibited by WS does not reduce the fault rate of WS below that of CD. For θ = 25 WS produces 85% more faults than CD, and ST_{WS} is higher than ST_{CD} by 762%. Together with the results in the previous subsection, this observation suggests that CD make better use of the allocated memory over execution time.

4.4.2. Space time cost

Minimizing the fault rate under WS by using large values of τ may produce high space time costs. Therefore, a more realistic cost measure of WS policy is the space time cost. In fact, WS is advocated as a near optimal policy in terms of minimizing space time costs. Moreover, ST has been used to control the system throughput. A maximum throughput is claimed to be achieved when the

Table 4-5 (SYSTEM: MPL=4, 5, 10)
CD compared with the minimal achievable page faults under WS
with corresponding space time costs

		ST Cost(10^7)			Page Faults			
MPL	θ	CD	WS	Δ_F %	WS	CD	Δ_{ST} %	τ
4	10	12.34	33.2	170%	31681	10987	188%	9
	50	4.77	31.6	562%	6858	5643	22%	116
	100	5.29	45.5	760%	4268	1391	207%	6200
	150	4.77	11.0	131%	1117	1309	-19%	6200
	200	4.77	11.7	145%	725	1309	-45%	20,000
5	50	12.05	30.4	152%	11875	6931	72%	31
	100	5.84	9.44	62%	2489	2081	20%	601
	150	5.09	11.8	132%	1303	1972	-34%	6500
	200	5.28	12.8	142%	991	1955	-50%	25,000
10	50	13.62	49.1	260%	29332	12174	141%	101
	100	13.54	88.1	551%	22870	4226	441%	551
	150	11.13	27.5	147%	6241	4080	53%	551
	200	12.17	22.7	87%	4055	3984	02%	3000

space time cost is minimized [12], [20]. In this subsection we compare minimal space time costs achieved under WS with those achieved under CD for different values of θ.

The results for individual processes are reported in Tables 4-6a-e for MPL=4, 5 and 10. For each process we find τ_{ST-min} which minimizes the space time cost of that process. The space time costs and the number of page faults generated using τ_{ST-min} are compared with the space time costs and number of page faults generated under CD. The relative difference between ST_{WS} and ST_{CD} is given by Δ_{ST} in Equation (4-2): Δ_F is given in Equation (4-1). Positive Δ_{ST} and Δ_F mean that WS has a higher space time cost and generates more faults than CD. The value Δ_F is used to study the time cost due to running each process at its minimal space time cost. A low space time cost may result from using relatively small memory at the expense of generating many faults.

Table 4-6a (MAIN)
CD compared with the minimal achievable space time cost under WS
with corresponding page faults (MPL=4, 5, 10)

		ST Cost(10^6)			Page Faults			
MPL	θ	CD	WS	Δ_{ST}%	CD	WS	Δ_F%	τ
4	10	3.84	7.59	98%	923	1469	59%	9
	20	3.84	5.87	53%	923	947	03%	10
	25	3.84	5.71	49%	923	921	00	6
	30	4.37	5.81	33%	889	921	04%	10
	40	4.33	5.71	32%	855	921	08%	6
	50	4.33	5.81	34%	855	921	08%	10
	100	5.21	5.81	11%	169	921	445%	10
	150	7.12	5.82	-18%	152	921	506%	15
	200	7.12	9.87	39%	152	920	506%	201
5	50	4.33	5.81	34%	855	921	08%	11
	100	4.72	5.86	24%	237	921	289%	51
	150	8.11	5.86	-28%	245	921	276%	51
	200	7.12	5.86	-18%	152	921	506%	51
10	50	4.33	6.11	41%	895	1029	15%	11
	100	3.824	5.96	56%	906	956	06%	51
	150	8.11	5.86	-28%	245	921	276%	51
	200	7.12	5.86	-18%	152	921	506%	51

Table 4-6b (FIELD)
CD compared with minimal achievable space time costs under WS
with corresponding page faults (MPL=4, 5, 10)

		ST Cost(10^6)			Page Faults			
MPL	θ	CD	WS	Δ_F %	CD	WS	Δ_{ST} %	τ
4	10	8.95	32.3	261%	2501	4546	261%	5
	20	2.85	21.7	660%	173	1806	944%	15
	25	2.90	3.14	08%	136	241	77%	21
	30	2.75	2.81	03%	136	193	42%	30
	40	2.90	3.11	08%	136	161	18%	21
	50	2.90	3.03	05%	136	166	22%	86
	100	2.90	3.07	06%	136	161	18%	20
	150	2.90	3.19	10%	136	72	-47%	24,000
	200	2.90	3.53	22%	136	97	-29%	1000
5	50	2.90	3.01	04%	136	214	57%	61
	100	2.90	3.14	09%	136	168	24%	151
	150	2.90	3.42	18%	136	72	-47%	3500
	200	2.90	3.52	22%	136	97	-29%	951
10	50	2.76	4.82	75%	135	405	200%	21
	100	2.85	3.07	08%	165	164	00	700
	150	2.89	3.25	12%	128	167	30%	201
	200	2.89	3.03	05%	128	83	-35%	1800

Tables 4-6a-e show that CD has considerably lower space time than WS. Consider program MAIN. For MPL=4, ST_{WS} is larger than ST_{CD} for all θ values except θ=150. However, for θ=150 WS generates 5 times more page faults than CD in order to achieve 18% less space time cost. Similarly, for MPL=5, ST_{WS} is larger than ST_{CD} for θ=50 and 100. Note that the low space time cost under CD is not achieved at the expense of a large number of page faults: for θ=100 ST_{CD} is 24% less than ST_{WS} and F_{CD} is almost 3 times less than F_{WS}. A low ST cost under CD is due to a relatively lower page fault number and a relatively lower memory consumption. In Table 4-6b, the results are shown for program FIELD. The space time cost under CD, ST_{CD}, is lower than ST_{WS} for all θ and MPL values. For θ=150 and 200, WS achieves a lower number of page faults than CD. For such large values of θ, WS can use a large τ value to generate a minimum number of faults. CD, however, achieves a minimum number of faults for much smaller θ values, e.g., θ=25 for MPL=4

Table 4-6c (INIT)
CD compared with minimal achievable space time costs under WS
with corresponding page faults (MPL=4, 5, 10)

		ST Cost(10^6)			Page Faults			
MPL	θ	CD	WS	Δ_{ST}%	CD	WS	Δ_F%	τ
5	50	5.04	17.8	253%	729	1016	39%	31
	100	9.28	10.7	15%	273	282	03%	601
	150	9.28	9.44	02%	273	178	-35%	501
	200	9.28	9.44	02%	273	178	-35%	501
10	50	11.74	44.0	275%	535	2634	392%	11
	100	10.7	16.7	56%	273	523	92%	551
	150	3.15	9.43	199%	273	215	-21%	601
	200	9.33	9.33	00	273	184	-33%	551

and θ=50 for MPL=5 and 10. Similarly for program INIT, WS achieves lower fault number than CD when CD achieves lower space time cost for θ=150, and 200 for MPL=5 and 10. Again this is because WS can generate a close to the minimal page fault number by using a relatively large τ. The virtual size of INIT is 175 pages; WS generates 178 pages for $\theta \geq 500$. The CD policy achieves 273 faults at its best. However, CD still has a lower space time cost than the minimal achievable under WS.

Tables 4-6 show that the space time cost of WS, when WS is properly tuned, is considerably larger than the space time cost of CD for most of the time. Even when WS has a lower space time cost, its page faults number is higher than CD's and the low ST is mainly due to small memory consumption. Our results show that WS is not optimal in terms of minimizing fault rate as claimed in [20]. However, CD remains to be compared with DMIN to show how close to optimal it can generate a space time cost.

In a multiprogramming system, minimizing the space time cost of individual processes may not serve the purpose of optimizing the system performance. It would have been very helpful if the processes in the system utilized one τ to achieve their minimal space time cost. Graham, and Denning [26] claim that all processes in the system can use one τ to achieve a space time cost within

Table 4-6d (CONDUCT)
CD compared with the minimal achievable space time cost under WS
with corresponding page faults (MPL=5, 10)

		ST Cost(10^6)			Page Faults			
MPL	θ	CD	WS	Δ_{ST} %	CD	WS	Δ_F %	τ
5	50	96.9	106.0	09%	4562	5144	13%	21
	100	30.17	37.7	25%	789	754	-04%	601
	150	22.22	30.17	36%	748	611	-18%	601
	200	22.22	30.17	36%	748	611	-18%	601
10	50	38.0	106.0	179%	3873	5144	33%	21
	100	30.17	37.7	25%	789	754	-04%	601
	150	22.22	30.17	36%	748	611	-18%	601
	200	22.22	33.70	52%	748	679	-09%	801

Table 4-6e (HWSCRT)
CD compared with the minimal achievable space time cost under WS
with corresponding page faults (MPL=5, 10)

		ST Cost(10^6)			Page Faults			
MPL	θ	CD	WS	Δ_{ST} %	CD	WS	Δ_F %	τ
5	50	11.33	23.10	104%	649	5766	788%	1
	100	11.33	19.50	72%	646	378	-41%	401
	150	11.33	13.3	18%	646	155	-76%	6500
	200	11.33	13.5	19%	646	188	-70%	30,000-
10	50	11.33	23.1	104%	649	5766	788%	1
	100	19.23	23.1	20%	649	5766	788%	1
	150	19.28	20.8	08%	646	486	-25%	401
	200	19.28	15.1	-22%	646	347	-46%	451

10% of the minimal space time cost (10% detuned policy). The goal is, therefore, to minimize the overall system space time cost, assuming that individual processes are within P% of their minimal ST values. In Table 4-7 the minimal system space time cost, ST_{sys}, under WS is compared with ST_{CD}.

For WS, we find a window size, $\tau_{sys-ST-min}$, which minimizes the overall system space time cost, $ST_{sys-min}$, which is compared with ST_{CD}. The number of page faults generated using $\tau_{sys-ST-min}$ is also found and compared with F_{CD}. Table 4-7 shows that CD outperforms WS by a

Table 4-7 (SYSTEM)
CD compared with minimal achievable space time costs under WS
with corresponding page faults (MPL=4, 5, 10)

		ST Cost(10^7)			page faults			
MPL	θ	CD	WS	Δ_F%	WS	CD	Δ_{ST}%	τ
4	10	12.34	33.2	170%	31681	10987	188%	9
	50	4.77	13.7	187%	7186	5643	27%	25
	100	5.29	13.7	159%	7186	1391	417%	25
	150	4.77	6.25	31%	1830	1309	40%	590
	200	4.77	6.33	32%	1830	1309	40%	601
5	50	12.05	28.1	133%	19313	6931	178%	11
	100	5.84	9.44	62%	2489	2081	20%	601
	150	5.09	8.06	58%	2149	1972	10%	601
	200	5.28	8.06	53%	2149	1955	10%	601
10	50	13.62	49.1	260%	29332	12174	141%	21
	100	13.54	84.6	525%	23885	4226	465%	51
	150	11.13	27.5	147%	6241	4080	53%	551
	200	12.17	17.8	46%	4710	3984	21%	601

great margin, especially for θ=10, 50, and 100. Note that the improvement of CD over WS increases with increasing MPL for the same θ values. For instance, for θ=150, CD outperforms WS by 31%, 58%, and 147% for MPL=4, 5, and 10, respectively. The CD policy achieves lower faults numbers than WS for all θ and MPL values exclusively. The negative Δ_{ST} and Δ_F values in Tables 4-6 disappear in Table 4-7, indicating that a process may have a lower space time cost under WS than ST under CD at the expense of some other process in the system.

The corresponding ST's and page faults for individual programs are found when the overall system ST is minimized, using $\tau_{SYS-ST-min}$. These values are compared with ST_{CD} and F_{CD} for the individual processes. The results are reported in Table 4-8 for MPL=3. Table 4-8 shows that CD outperforms WS at the individual process level when the overall system performance is being optimized.

Table 4-8
Optimizing system performance, MPL=3

	Δ_F %				Δ_{ST} %			
θ	MAIN	FIELD	INIT	System	MAIN	FIELD	INIT	System
6	85	2464	35	164	113	733	136	215
7	65	2094	23	133	96	1058	123	248
8	45	1956	21	119	83	1048	130	248
9	16	1864	4	95	59	1072	106	232
10	6	1840	1	91	50	1093	108	235
11	5	1836	00	89	48	1090	109	235
12	24	2016	3	89	66	1356	120	362
13	28	2110	3	91	72	1345	228	361
14	17	2041	2	85	60	1363	236	367
15	5	1930	2	77	53	1407	246	328
16	4	1921	2	77	52	1407	249	380
17	20	1163	2	52	64	869	340	365
18	6	1151	2	48	54	1114	356	408
20	7	1136	2	47	54	1176	363	384
25	9	1076	166	85	33	1269	1111	762
30	8	18	6	7	35	11	200	98
35	8	18	6	7	35	14	16	31
40	9	29	1	6	53	20	138	104
45	8	18	2	5	52	24	124	95
50	8	21	157	50	54	19	288	191
100	131	-48	-27	15	377	18	31	106

4.4.3. System throughput

A major design goal in a multiprogramming system is to maximize the number of jobs completed per unit time, i.e., the system throughput (Φ). In Table 4-9, the maximum throughput achieved under WS (Φ_{WS}) is compared with the throughput under CD (Φ_{CD}) for MPL=3,5,10. The relative difference (Δ_Φ) between the WS's maximum throughput and CD's throughput is given by

$$\Delta_\Phi = \frac{\Phi_{CD} - \Phi_{WS}}{\Phi_{WS}} \times 100\% \quad . \tag{4-3}$$

Table 4-9 shows that CD outperforms WS by a large margin, especially for smaller values of θ. Consider, for example, MPL=3. For θ = 6, CD has a higher throughput than WS by a factor of ~ 15. For θ = 100, CD achieves a 13% higher throughput than WS. For MPL=10, CD achieves higher

throughput than WS by 87% and 27% for θ=100 and 150, respectively. The results suggest that CD outperforms WS when the memory is highly utilized.

Table 4-9
CD compared with the maximum achievable throughput under WS

	Throughput Φ (10^{-7})			
MPL	θ	CD	WS	Δ_Φ
3	6	4.09	0.26	1497%
	7	4.09	1.77	131%
	8	4.09	1.88	118%
	9	4.09	2.11	94%
	10	4.09	2.16	89%
	11	4.09	2.18	88%
	12	4.21	2.23	89%
	13	4.21	2.22	90%
	14	4.21	2.29	84%
	15	4.21	2.39	76%
	16	4.21	2.39	76%
	17	4.21	2.79	51%
	18	4.21	2.86	47%
	20	4.27	2.89	48%
	25	7.53	4.11	83%
	30	7.55	7.05	7%
	35	7.55	7.09	6%
	40	7.55	7.12	6%
	45	7.55	7.17	5%
	50	10.6	7.20	47%
	100	23.9	21.1	13%
4	10	1.81	0.63	187%
	20	2.35	2.15	09%
	25	2.41	2.23	08%
	50	2.67	2.88	-7%
	100	13.5	4.63	192%
	150	14.3	16.7	-14%
	200	14.3	24.5	-42%
10	50	1.83	1.69	09%
	100	4.05	2.17	87%
	150	8.89	7.76	28%
	200	11.5	11.7	00

4.4.4. Controllability

In a multiprogramming system it is necessary to tune WS policy in order to find a suitable τ to achieve a desired goal. For CD this problem does not exist since the directives are inserted at compile time and executed as part of the code at run time. Memory allocation is performed dynamically as the directives are received by the operating system. Finding the appropriate τ in the WS case can be a tedious problem for several reasons.

The first reason is the anomalous behavior of WS's fault rate function discussed in Chapter 2. With the the existence of anomalies, fault rate reduction is not always achievable by increasing τ. Instead, the fault rate may increase. Moreover, the fault rate anomalies distort the shape of fault rate function curves and, hence, the lifetime curves. Life time curves because of the anomalies do not exhibit well defined knees; knees in a lifetime curve are essential for the primary knee criterion [20]. The primary knee criterion suggests that the primary knee of a lifetime curve is approximately associated with the minimum space time cost point; i.e., by using τ where the primary knee occurs a process would be running with minimal space time cost. For this reason, Denning rejects lifetime models of program behavior if they do not exhibit knees [20]. With the existence of τ-F anomalies, it is not obvious how one would locate the knees of a lifetime curve. The primary knee criterion, therefore, may not be useful for controlling WS.

The second reason is the difficulty of controlling the policy to produce the maximum possible throughput. It has been assumed [12] that the maximum throughput is achieved by minimizing the space time cost. The average ST of a process in the system is given by

$$ST = \frac{\theta \times T}{N} \quad . \tag{4-4}$$

where N is the number of jobs in the system and T is the total elapsed time of all the jobs in the system. The throughput, Φ, is given by

$$\Phi = \frac{N}{T} \quad \text{and so} \quad ST = \frac{\theta \times T}{\Phi \times T} \quad . \tag{4-5}$$

Equation 4-5 implies that a maximum throughput can be achieved if each process in the system

operates at its minimal space time point, or, equivalently, minimizing the overall system space time cost. This argument is not realistic for two reasons. First, the above formula assumes that the memory space, θ, is completely utilized. This assumption is not always true, especially for large values of θ.

The second reason is that minimizing the space time cost of each process does not necessarily minimize the overall system ST. Each process may have its own optimal τ which differs from those used by other processes in the system. The assumption that the space time cost has a flat minimal region, meaning that a wide range of τ can minimize the space time cost, has been shown to be optimistic for individual programs running in a single programming machine [3], [6], [8].

Our results also show that each program may use a different optimal τ. For example, for MPL=3 and θ = 50, three values of τ (τ = 6, 316, 7000) are needed to minimize the ST of MAIN, FIELD and INIT, respectively. Tables 4-6 further illustrate this fact. For example, for MPL=5 and θ=150, five values of τ (51, 501, 601, 3500, 6500) are used by programs MAIN, INIT, CONDUCT, FIELD, HWSCRT, respectively. Similarly, for θ=200 and MPL=200, five values of τ are used (51, 451, 551, 601, 1800). Furthermore, a process using an optimal τ, $\tau_{sys-opt}$ from the system's stand point, may run with a relatively large space time cost compared to its local minimum space time cost with τ_{min}. In Table 4-10 we show for each program both space time cost values $ST(\tau_{sys-opt})$ and $ST(\tau_{min})$. The relative difference between these values is given by

$$\Delta = \frac{ST(\tau_{sys-opt}) - ST(\tau_{min})}{ST_{min}} \times 100\% \ .$$

In Table 4-10 the optimal τ for which the system space time cost is minimized is 601 for MPL=10 and θ=200. For the moment we assume that it is possible to find an optimal τ value which minimizes the space time cost of each process or the space time cost of all the processes in the system. The question is whether using this τ achieves a maximum throughput; i.e., is this an optimal τ?. Equation (4-5)[12], [20] gives a positive answer to this question. We have argued that this is true only if the memory is completely utilized. Our results show that for underutilized memory the real maximum throughput can deviate from the throughput achieved by using τ which minimizes the space

Table 4-10
Relative difference between global ST and local ST for each process

	MPL=10; θ=200; τ_{opt} =601			
Program	τ_{min}	$ST_{min}(10^6)$	$ST_{opt}(10^6)$	Δ
MAIN	51	5.86	19.5	233%
FIELD	1800	3.03	3.42	13%
INIT	551	9.33	9.71	04%
HWSCRT	801	33.7	40.1	20%
CONDUCT	451	15.1	18.9	25%

time cost by almost a factor of 2. In Table 4-11 we show the relation between the maximum throughput Φ_{max} and the throughput achieved at the minimum space time cost point $\Phi_{ST_{min}}$. The relative difference between these two values is given by

$$\Delta = \frac{\Phi_{max} - \Phi_{ST_{min}}}{\Phi_{ST_{min}}} \times 100\% \ .$$

Table 4-11 shows that for relatively small memory sizes the minimum space time cost and the maximum throughput are achieved by using the same τ. See in Table 4-11 the entries for θ=6, 10 for MPL=3; θ=10, 20, 30, 40 for MPL=4; θ=100 for MPL=5; and θ=150 for MPL=10. However, for larger values of θ, Φ_{max} deviates from Φ_{ST-min} by a large percentage. For example, Δ =167% for θ=100 and MPL=3; For θ=200, Δ=136%, 106%, 15% for MPL=4, 5, and 10, respectively. It is worthwhile to mention, however, that WS has a poor performance compared to CD when the memory is highly utilized: small values of θ in Tables 4-2, 4-3, 4-4.

4.5. Summary and Conclusions

We have presented in this chapter performance measurements on program behavior in multiprogramming systems. Program traces are simulated in a multiprogramming system under CD, a compiler directed memory management policy, and WS, a dynamic policy. We have compared the performance of CD with that of WS since the latter has been claimed [20] to outperform other existing policies. Four characteristics of multiprogramming virtual memory systems have been investigated: page faults, space time cost, system throughput, and controllability.

Table 4-11
Maximum throughput versus throughput at ST_{min} under WS

		Throughput Φ (10^{-7})		
MPL	θ	Φ_{max}	$\Phi_{ST_{min}}$	Δ_Φ%
3	6	0.26	0.26	00
	10	2.16	2.16	00
	17	2.79	2.2	27
	18	2.86	2.2	30
	20	2.89	2.2	31
	25	4.11	2.2	87
	30	7.05	7.05	00
	40	7.12	7.05	1
	50	7.20	7.05	2
	100	21.1	7.05	201
	200	28.6	10.7	167
4	10	.63	.63	00
	20	1.77	1.77	00
	25	2.23	1.84	22%
	30	2.70	2.68	00
	40	2.76	2.75	00
	50	2.88	2.75	05%
	100	4.63	2.75	68%
	150	16.7	10.4	61%
	200	24.5	10.4	136%
5	50	2.09	1.29	62%
	100	9.65	9.65	00
	150	17.8	11.1	61%
	200	22.9	11.1	106%
10	50	1.69	1.69	00
	100	2.17	2.08	04%
	150	7.76	7.76	00
	200	11.7	10.2	15%

The results reported in this chapter show that CD outperforms WS by a fairly large margin, especially when the memory is highly utilized. CD is able to dynamically allocate memory space according to the need of a running program, the available memory space, and the need of other processes in the system. The outcome of this facility is a relatively low fault rate at a relatively low memory space cost and, hence, a low space time cost. More importantly, CD is shown to have a

higher throughput than WS.

We have also illustrated that WS lacks controllability while CD does not exhibit controllability problems at all. CD is a parameterless policy while WS has a parameter, τ, which needs to be tuned in order to achieve a desired goal. It is necessary, for instance, to find τ that minimizes the space time cost in order to maximize system throughput [20]. However, it is not obvious how one would choose the right τ to minimize ST, even using the primary knee criterion [1], [12]. The primary knee criterion is difficult to apply due to τ-F anomalies exhibited by WS. See Chapter 2. In any case, we showed that using an optimal τ which minimizes the space time cost does not, necessarily, maximize the throughput. ST_{min} maximizes the throughput only when the memory is completely utilized, but then WS has a poor performance compared to CD.

CHAPTER 5

CONCLUSIONS

5.1. Summary of Results

A new approach to the management of numerical programs in virtual memory systems is presented in this study. We have presented a compiler directed policy (CD) which incorporates two memory directives: 1) ALLOCATE and 2) LOCK and UNLOCK. ALLOCATE estimates the memory requirements of a process at compile time. Memory requirements are passed to the operating system at run time through two primitives: the amount of memory requested and the priority of the request. The CD plicy is designed to dynamically adjust a program's memory allocation according to the status of the available free memory on the system which dynamically changes as processes acquire and release memory space. For this purpose, CD incorporates a swapping mechanism. Subprogram control structures are handled dynamically at run time, thus enabling the preprocessor at compile time to consider each subroutine as a whole unit.

The performance of CD is evaluated using a trace driven simulator of a multiprogramming system. Traces of numerical programs are used in the experiments. The performance of CD is compared to the performance of WS policy. The results reported in Chapter 4 show that CD is superior to WS in high memory contention cases. The CD policy produces lower fault rates and lower space time costs than WS, and therefore, achieves higher throughput. As a result, CD is able to support higher multiprogramming levels for a given size of physical memory.

We have presented evidence in this study, that WS has a controllability drawback. In Chapter 2, we reported empirical results on the WS anomalies. The anomaly types exhibited by WS are related directly to the WS control parameter, the window size τ. Thus, tuning WS to achieve a desired performance is not always attainable because of the anomalous behavior. The anomaly types reported in this thesis are not exhibited by WS when tested in a uniprogramming

environment. The results suggest that conclusions based on experiments with individual single programs should not be used in a simplistic manner in multiprogramming systems. It has also been observed that it is not possible to find a single value for the control parameter which can be used by every process in the system.

On the other hand, CD exhibits no controllability problems and has no control parameter. Memory requests issued upon executing a directive are processed by the operating system, granted or rejected, according to the available free memory.

In conclusion, this thesis has

(1) presented CD, a compiler directed memory management policy for numerical programs,

(2) shown that WS exhibits anomalies in multiprogramming systems, otherwise unpredicted from experiments with uniprogramming systems,

(3) shown that CD outperforms WS by a relatively large margin.

5.2. Suggestions for Future Research

The compiler directed policy presented in this thesis applies only to numerical programs. The extension of CD to other program categories is essential before such an approach to the memory management problem can be adopted. The locality characteristics of different application programs have to be understood thoroughly before memory directives can be designed. Typical applications are data base systems, system programs, and AI application programs. The compiler directed policy is designed for single processor machines. However, the ideas used in this thesis can be useful in pursuing similar techniques in multiprocessing systems.

The performance of CD compared to WS, although the latter is claimed to be the best nonlookahead policy [20], is not sufficient to evaluate the performance of CD, which should be compared to other dynamic policies such as PFF, global LRU, and global CLOCK. Moreover, it is essential to evaluate the performance of CD when comparing it with the optimal policies. For instance, CD should be compared with DMIN [10], which generates the absolute minimum space time cost.

Also, we feel that performance evaluation techniques for virtual memory systems should be upgraded to include multiprogramming specific characteristics. For instance, one should be able to measure the influence of one program on the rest of the programs in the system. This is necessary for scheduling strategies. The techniques developed in this study can also be used to enhance scheduling strategies, especially in allocating time slots to running processes.

Finally, the main issue which remains to be pursued is the issue of implementation. The complexity of such a problem lies in the fact that CD has to be incorporated into both the compiler and the operating system. Furthermore, some architectural features are necessary to implement CD, particularly at the processing stage of a directive. Therefore, an integrated approach to the design of computer systems is necessary for CD to be implemented in real systems.

REFERENCES

[1] M. Abaza, "On the Effectiveness of Memory Management System Calls In VAX/VMS," M.S. Thesis, Yarmouk University, Dep. Elect. Eng., October 1984.

[2] W. Abu-Sufah, D. Kuck, and D. H. Lawrie, "Automatic Transformations for Virtual Memory Computers," *Proc. of the 1979 National Computer Conf.*, pp. 969-974, June 1979.

[3] W. Abu-Sufah, R. Lee and M. Malkawi, "Identifying Two Program Categories for Memory Management Purposes," *Proc. of the 1984 IEEE 8th International COPMSAC*, pp. 492-503, November 1984.

[4] W. Abu-Sufah and D. A. Padua, "Some Results on the Working Set Anomalies in Numerical Programs," *IEEE Trans. on Software Engineering*, vol. SE-8, no. 2, pp. 97-106, March 1982.

[5] W. Abu-Sufah, "Identifying Program Localities at the Source Level," University of Illinois, Dep. Comp. Science, Rep. No. UIUCDCS-R-82-1108, October 1982.

[6] T. O. Alanko, H. J. Haikala, and P. H. Kutvonen, "Methodology and Empirical Results of Program Behavior Measurements," *Performance 80, ACM Sigmetrics Performance Evaluation Review*, vol. 9, no. 2, pp. 55-66, Summer 1980.

[7] W. Abu-Sufah, R. Lee, M. Malkawi, and P-C. Yew, "Empirical Results on the Behavior of Numerical Programs in Virtual Memory Systems," University of Illinois, Dep. Comp. Science, Report No. UIUCDCS-R-81-1076, November 1981.

[8] W. Abu-Sufah, R. Lee, M. Malkawi, and P-C. Yew, "Experimental Results on the Paging Behavior of Numerical Programs," *Proc. of the 6th International Conf. on Software Engineering*, pp. 110–117, September 1982.

[9] A. P. Batson, D. W. E. Blatt, and J. P. Kearns, "Structure Within Locality Intervals," in *Measuring, Modelling and Evaluating Computer Systems*, H. Beilner and E. Gelenbe, Eds., Amesterdam, The Netherlands: North-Holland, 1977.

[10] R. Budzinski, E. Davidson, W. Mayeda, and H. Stone, "DMIN: An Algorithm for Computing the Optimal Dynamic Allocation in a Virtual Memory Computer," *IEEE Trans. on Software Engineering*, vol. SE-7, no. 1, pp. 113-121, January 1981.

[11] R. Budzinski, E. Davidson ,"A Comparison of Synamic and Static Virtual Memory Allocation Algorithms," *IEEE Trans. on Software Engineering* vol. SE-7, no. 1, pp. 122-131, January 1981.

[12] J. P. Buzen, "Optimizing the Degree of Multiprogramming in Demand Paging Systems," *Proc. IEEE COMPCON*, pp. 139-140, September 1971.

[13] R. W. Carr, "Virtual Memory Management", Ph. D. Thesis, Computer Science Department, Stanford University, August 1981.

[14] W. W. Chu and H. Opderbeck, "The Page Fault Frequency Replacement Algorithm," in *1972 AFIPS Conf. Proc., Fall Joint Comput. Conf.*, vol. 41, AFIPS Press, pp. 597-609, 1972.

[15] W. W. Chu and H. Opderbeck, "Program Behavior and the Page Fault Frequency Replacement Algorithm," *Computer*, vol. 9, no. 11, pp. 29-38, November 1976.

[16] P. J. Denning and G. S. Graham, "Multiprogramming Memory Management," *IEEE. Proc.*, vol. 63, pp. 924-939, June 1975.

[17] P. J. Denning and K. C. Kahn "A Study of Program Locality and Life-time Functions," *Proc. 5th Symp. Operating Systems Principles, ACM SIGOPS*, pp. 207-216, November 1975.

[18] P. J. Denning, "Working Set Model for Program Behavior," *Comm. of the ACM*, vol. 11, no. 5, pp. 323-333, May 1968.

[19] P. J. Denning, "On Modeling Program Behavior," *Proc. AFIPS SJCC*, pp. 937-945, 1972.

[20] P. J. Denning, "Working Sets Past and Present," *IEEE Trans. on Software Eng.* vol. SE-6, no. 1, pp. 64-84, January 1980.

[21] D. Ferrari, "Improving Locality by Critical Working Sets," *Comm. ACM vol. 17, pp. 614-620, November 1974.*

[22] D. Ferrari, "Considerations on the Insularity of Performance Evaluation," IEEE Trans. on Software Engineering, vol. SE-12, no. 6, pp. 678-683, June 1986

[23] D. Ferrari and Y-Y. Yih, "VSWS: The Variable-Interval Sampled Working Set Policy," *IEEE Trans. on Software Engineering*, vol. SE-9, no. 3, May 1983.

[24] M. A. Franklin, G. S. Graham, and R. K. Gupta, "Anomalies with Variable Partition Paging Algorithms," *Comm. of the ACM*, vol. 21, no. 3, pp. 232-236, March 1978.

[25] G. S. Graham, "A Study of Program and Memory Policy Behavior," Ph.D. thesis, Purdue University, Dep. Comp. Science, December 1976.

[26] G. S. Graham and P. J. Denning, "On the Relative Controllability of Memory Policies," in *Computer Performance*, K. M. Chandy and M. Reiser, Eds., Amsterdam, The Netherlands: North-Holland, pp. 411-428, August 1977.

[27] R. B. Hagman and R. S. Fabry, "Program Page Reference Patterns," *Proc. of the 1982 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 20-29, August 1982.

[28] H. J. Haikala and H. Pohijanlahti, "On the BLI-Model of Program Behavior," *Proc. of the 1983 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 28-38, August 1983.

[29] J. Kearns and S. DeFazio, " Locality of Reference in Hierarchical Database Systems," *IEEE Trans. on Software Eng.*, vol.SE-9, no. 2, March 1983.

[30] A. W. Madison and A. P. Batson, " Characteristics of Program Localities," *Comm. of the ACM*, vol. 19, no. 5, pp. 285-294, May 1976.

[31] Mohammad Malkawi, "Some Aspects of Numerical Program Behavior In Virtual Memory Systems," M.S. Thesis, Dep. Elect. Eng., Yarmouk University, Jordan, June 1983

[32] J. B. Morris, "Demand Paging Through the Use of Working Sets on the MANIAC II," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 867-872, October 1972.

[33] B. G. Prieve and R. S. Fabry, "VMIN: An Optimal Variable Space Page Replacement Algorithm," *Comm. of the ACM*, vol. 19, no. 5, pp. 295-297, May 1976.

[34] J. Rodriguez-Rosell and J. P. Dupuy, "The Design, Implementation and Evaluation of a Working Set Dispatcher," *Commun. of the ACM*, vol. 16, pp. 556-560, September 1973.

[35] R. Simon, "The Modeling of Virtual Memory Systems," Ph.D. Thesis, Purdue University, Dep. Comp. Science, September 1979.

[36] A. J. Smith, "A Modified Working Set Paging Algorithm," *IEEE Trans. on Computers*, vol. C-25, no. 9, pp. 907-914, September 1976.

[37] S. S. Thakkar, and A. E. Knowles, "A High Performance Memory Management Scheme," *IEEE, Computer*, pp. 8-20, May 1986.

[38] I. L. Traiger, "Virtual Memory Management for Database Systems," *SIGOPS Symp. on OS Review*, pp. 26-48, October 1982.

[39] K. S. Trivedi, "Prepaging and Application to Array Algorithms," *IEEE Trans. Compt., vol. 12, no. 4, pp. 39-56, 1978.*

[40] A. I. Verkamo, "Empirical Results on Locality in Database Referencing," *Proceedings of the 1985 ACM SIGMETRICS Conf. on Measurement and Modeling*, pp. 49-58, May 1985.

VITA

Mohammad Isam Malkawi was born on September 15, 1957 in Jordan. He was an American Field Service foreign exchange student in 1972-1973 and graduated from Arcola High School, Arcola, Illinois. In 1974 he received the Jordanian general secondary school certificate from Irbid Secondary School, Jordan. He then won a scholarship to study computer engineering in the Soviet Union. In 1980, Malkawi graduated from Tashkent Polytechnical Institute with honors. From 1980 until 1983 he attended Yarmouk University in Jordan, where he received his master's degree in Electrical Engineering. During his stay at Yarmouk, he was a teaching and research assistant.

In 1983, Mohammad Malkawi joined the computer systems group at the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign as a graduate student and research assistant.

END

10-86

DTIC